DESIGN OF ON-LINE DECIMAL MULTIPLIER

BY

ABDULAZIZ SALAH TABAKH

A Thesis Presented to the

DEANSHIP OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

COMPUTER ENGINEERING

DATE : JUNE 2011

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN 31261, SAUDI ARABIA

DEANSHIP OF GRADUATE STUDIES

This thesis, written by ABDULAZIZ SALAH TABAKH under the direction of his thesis advisor and approved by his thesis committee, has been presented to and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of MASTER OF SCIENCE in COMPUTER ENGINEERING.

Thesis Committee

Dr. Alaaeldin A. Amin
Thesis Advisor

Dr. Aiman Elmaleh
Member

Dr. Abdulhafid Bouharaoua
Member

Dr. Basem M. Almadani
Department Chairman

Dr. Salam Zummo
Dean of Graduate Studies

26/6/11
Date

Dedication

To my father soul .. who inspired me to success.

To my beloved mother .. for her love and support.

To my sisters ...

Acknowledgments

All praise and thanks be to Almighty Allah, the most Gracious, the most Merciful. Peace and mercy be upon His Prophet.

Acknowledgment is due to the King Fahd University of Petroleum & Minerals (KFUPM) for supporting this research.

I am eternally grateful to my supervisor Dr. Alaaeldin A. Amin for his constant support and many suggestions without which I would not be able to successfully complete this research work. Many thanks are extended to my committee members Dr. Aiman Elmaleh and Dr. Abdulhafid Bouharaoua for their unlimited encouragement and useful suggestions.

I express my gratitude to Dr. Basem M. Almadani, Chairman of Computer Engineering Department for his support.

I am thankful for all the faculty and staff members of Computer Engineering Department for their support.

I would like to acknowledge my colleagues and friends for their encouragement.

Finally, I am grateful to my parents and family for their moral support, encouragement and prayers without which this work would not have been possible.

Abdulaziz S. Tabakh.

JUNE, 2011.

Contents

List of Tables

List of Figures

THESIS ABSTRACT

Name: ABDULAZIZ SALAH TABAKH

Title: DESIGN OF ONLINE
DECIMAL MULTIPLIER

Major Field: COMPUTER ENGINEERING

Date of Degree: JUNE 2011

Digit-serial online arithmetic schemes are highly attractive since they allow successive processes to be computed in an overlapping manner. This results in high throughput operation with simple communication interface and reduced area overhead. Decimal Arithmetic has an increasing importance and need specially in floating point operations. IEEE 854-2008 standard for decimal floating point operations was approved in June 2008. Since then, decimal arithmetic attracts researchers attention. In floating point operations, decimal arithmetic is more accurate over binary arithmetic. Simple floating point numbers (like 0.1, 0.2, and 0.3) need an infinite precision in binary while they can be represented accurately in decimal. Many programing languages (like Java, C/C++, COBOL, and others) provide decimal floating-point support that occupies around 50-90% of their processing time. A extensive study of decimal addition and multiplication algorithms is conducted and presented in this thesis. This study shows that there has been no reported algorithms or hardware implementations for an online decimal multiplication unit to date.

In this thesis, a design of online decimal multiplier is proposed. The design has two different architectures. The performance of the proposed design with both architectures is studied and compared. Both architectures are modeled using VHDL and synthesized for area and delay optimization.

MASTER OF SCIENCE DEGREE

King Fhad University of Petroleum and Minerals, Dhahran.
JUNE 2011

خلاصة الرسالة

الإســـــم : عبدالعزيز صلاح طباخ

عنوان الرسالة : تصميم ودراسة أداء اجهزة عملية الضرب التسلسلية (online)

التخصص : هندسة الحاسب الآلي

تاريخ التخرج : رجب 1432

الأساليب العددية الرقمية المتسلسلة (online) شائعة جداً نظراً لكونها تسمح بحساب العمليـــات المتتابعـــة بشكل متزامن. مما يؤدي إلى ارتفاع الكفاءة الإنتاجية وتقليل المساحة وزيادة كفاءة التواصل بين المكونات المختلفـــة. العمليات الرياضية في النظام العشري بدأت تستقطب الاهتمام بشكل متزايد خصوصاً لعمليات الاعداد الحقيقية بسبب الحاجة لهذا النظام. منظمة IEEE أقرت النظام المعياري IEEE 754-2008 للعمليـــات الحســـابية في النظـــام العشري في يونيو 2008 ومنذ ذلك الحين استقطب مجال الحساب العشري في اجهزة الحاسب الالي اهتمام البـــاحثين. تبرز الحاجة الى النظام العشري في الحاسبات في أن تمثيل الارقام الكسرية البسيطة مثل (0.1 و 0.2 و 0.3 على سبيل المثال) يحتاج الى عدد غير منتهي من الارقام الثنائية (bits) لذا تقوم اجهزة الحاسب بتقريبها حسب المساحة المتاحـــة في حين أنه يمكن تمثيلها بسهولة في النظام العشري ولهذا السبب عمد مطوروا لغات البرمجة (مثل لغـــات C,C++, Java, COBOL) الى تطوير واجهات برمجية لتمثيل الارقام الكسرية في النظام العشري. هذا الدعم من مطـــوري لغات البرمجة يجعل البرامج المطورة بهذة اللغات تستهلك ما بين 50-90% من وقت البرنـــامج في معالجـــة الارقـــام العشرية.

حتى اليوم، تشير التقارير إلى عدم تواجد أي خوارزميات أو أجهزة تقدم حلاً لوحدة الضـــرب المتسلســـل (Online) في النظام العشري. هذه الرسالة تقدم تصميم مقترحاً لوحدة ضرب متسلسل في النظام العشري. وقـــد تم عمل مقارنات لأداء هذا التصميم مع الأساليب التقليدية لعمليات الضرب. بالإضافة إلى ذلك، تم ادراج نتائج التركيب (synthesis) لمقارنة المساحة والتأخيرالزمني لهذه الخوارزميات.

CHAPTER 1

INTRODUCTION AND MOTIVATION

1.1 Introduction

In this section, the motivation for research in decimal arithmetic - particularly decimal multiplication - is presented. The inaccuracy of binary computer arithmetic in floating-point calculations and inefficient software solutions for decimal arithmetic will be highlighted.

Most current computer systems today have floating-point arithmetic units based on binary standards. This is not only because binary data can be efficiently stored in digital machines [5] but also because they are easy to implement in hardware [6]. In spite of their advantages, there are reasons that push towards having decimal

floating-point arithmetic. Most of data stored and manipulated in database systems are represented in decimal format which is affine to human nature. A survey was conducted on commercial databases of 51 major organizations covering a wide range of applications, including airline systems, banking, financial analysis, insurance, inventory control, management reporting, marketing services, order entry and processing, pharmaceutical, and retail sales, has shown that 55% of the numerical data were decimal and 43.7% of the remaining data were integers which can also be represented as decimals [7]. The primary motivation for decimal computer arithmetic is to enable users of computing systems to represent fractional data correctly and perform fractional decimal operations without representation errors, conversion errors, and rounding errors [4]. Commonly used fractional decimal numbers, like 0.1, 0.2, 0.3 etc, require infinite binary representation to be accurately represented. Instead, these fractions are rounded to fit in the available system precision. Accordingly, even if the subsequent operations are exact, the result is not exact [7]. This kind of approximation can be avoided if the numbers are represented in decimal format. Let us consider an example of sales tax calculation of 5% such as a $0.70 telephone call. Using binary floating-point, multiplying 0.70 by 1.05 will yield a result that is around 0.734999999999 which will be rounded to $0.73 that is less than the expected value obtained by decimal arithmetic ($0.735 rounded to $0.74)[7]. Such one-cent errors will add up in many cases leading to huge errors. For example, for a mobile telecom company serving millions of calls a day, the annual

loss could be in the order of millions. Thus, this minor approximation can cause a huge loss for the telecom company in the long term. Notice that this error is due to lack of accuracy in binary floating-point representation not due to rounding errors. For these reasons, financial calculations are preferably performed in decimal representation. Many programming languages support decimal floating-point arithmetic like C/C++, COBOL, Eiffel, Java, Lua, PERL,Python, Rexx, and Ruby [4]. Decimal data is usually represented as an integer scaled (divided) by a power of ten. Initial tests have shown that processing-bound applications (rather than I/O-bound applications) spend around 50-90% of their processing time in decimal processing suffering between 100× to 1000× performance penalty over hardware [7]. Thus, hardware solutions may enhance the execution speed of such applications by two to three orders of magnitude.

Due to the increase of decimal arithmetic importance in computer systems and the need for unified standard, IEEE approves a standard for decimal floating-point operations in June 2008 [9].

1.2 Thesis Objective and Organization

The purpose of this thesis is to design and implement a design for an online decimal multiplier. In this thesis work, the algorithm for online multiplication is introduced then it will be adopted for decimal multiplication operation. Online multiplication

algorithm accepts inputs from most significant digits and produces output in the same manner. Also the thesis includes a survey about decimal addition algorithms - specially signed-digit decimal addition - in order to adopt an encoding for decimal digits to optimize the speed and area of the implementation.

The objectives of the thesis are: (1) to study online multiplication operation, (2) to devise an online algorithm for decimal multiplication, (3) to design and implement an architecture for an online decimal multiplier, (4) model the proposed design using a Hardware Description Language (VHDL), (5) synthesize the VHDL code for area and delay in FPGA and evaluate performance and efficiency.

The rest of the thesis is organized as follows. In Chapter 2, background about decimal arithmetic is introduced. It includes a literature review about decimal multiplication, decimal addition, and online arithmetic, and summary about decimal section in the IEEE 754-2008 standard for floating-point. Chapter 3 provides detailed description about the proposed algorithm and its hardware implementation. This includes the selection of a digit set, various components in the design, and the details of the data path. Chapter 4 presents the synthesis results of the design. The designs are synthesized to optimize area and delay and these results are discussed and compared. Conclusions and future work are presented in Chapter 5.

CHAPTER 2

BACKGROUND

As this thesis discusses algorithms and designs for decimal multiplication, it is appropriate to present the standard of decimal floating-point (DFP) arithmetic and show current processor support for decimal arithmetic. The first section presents some of the existing hardware support for decimal arithmetic in modern computer systems. The second section of this chapter discusses the decimal multiplication operation, its phases, and possible alternatives. Decimal addition is investigated with different approaches and digit sets in the third section. A summary about the specifications and format of decimal arithmetic included in the IEEE 754-2008 standard [9], which constitutes the basis of all modern DFP (decimal floating-point) arithmetic is presented in the fourth section. The last section details the serial operations in computer arithmetic, highlights the online arithmetic, and focuses on online multiplication.

2.1 Hardware Support in Modern Systems

Since the IEEE 754-2008 standard has been approved in June 2008 [9], few DFP units that are compliant with the standard have made it to the market [1]. The first IEEE 754-2008 compliant hardware has been reported by IBM Power6 dual-core processor released in June 2007 and IBM z10 mainframe quad-core microprocessor released in March 2008 [1]. Other manufacturers, such as Intel, have software implementations but they are planning to either incorporate DFP units or to add hardware support for IEEE 754-2008 decimal operations [1].

2.2 Decimal Multiplication

Multiplication operation consists of three essential stages (phases): generation of partial products, reduction of partial products into two operands representing the sum in redundant form and final conversion (usually using carry-propagate addition) to convert the result into non-redundant representation [2] [1]. To enhance the reduction operation, some implementations use recoding of the input operands in which case two recoding stages are used; one at the beginning and one at the end [4]. Decimal multiplication differs from binary multiplication in two aspects: first, the wide range of decimal digits [0,9] and second, is the low representation efficiency of decimal values in binary systems using regular binary-coded decimal (BCD8421)

where only 10 out of 16 possible representations are used [1]. The first aspect complicates the generation of multiples of the multiplicand where the multiplication unit should deal with 9 possible multiples compared to binary which dose not need more than the multiplicand itself. The second aspect adds more complexity to the generation and reduction of partial products [2]. Thus, decimal multiplication algorithms and hardware designs are more complex than their binary counterparts. Commercial hardware implementations of decimal fixed-point integer multipliers are based on iterative algorithms characterized by low performance (long execution time) and reduced area cost [1]. Academic research proposed sequential multipliers that improve the latency of commercial ones. Recently, parallel fixed-point decimal multipliers have been proposed [1].

2.2.1 Generation of Partial Products

Multiplication performs the operation $P = A \times B$ where A is the multiplicand, B is the multiplier, and P is the product (multiplication result). Both operands A and B are n-digit numbers while P is $2n$-digits in general [11]. In most hardware designs, these quantities are considered as unsigned numbers and the sign of the product - when needed - is determined by special logic circuit based on signs of inputs A and B. A straightforward approach is through a digit-by-digit iteration over the multiplier B starting with the least (most) significant digit each time adding the appropriate multiple of the multiplicand A to the partial result PP followed by a

one digit left (right)-shift of the result, which corresponds to multiplication (division) by 10 [11]. One approach suggests that all multiples should be calculated and stored in advance (before the beginning of the multiplication process) to reduce the delay. The set of multiples that include all multiples is called the *primary set* or *primary multiples* [11]. The most notable drawbacks of this approach is the significant delay of pre-calculations and the area needed to store these multiples [11]. The recurrence equation used by this approach is given by:

$$P_{i+1} = (P_i + r^n \times A \times b_i) \times 10^{-1}$$

where b_i is the i^{th} digit of the multiplier starting from the least significant digit, $P_0 = 0$, $0 \leq i < n$, and n is the number of digits. An alternative to generating and storing all multiples is to generate and store a reduced set of multiples called *secondary set*. The main characteristic of this reduced set is that all remaining multiples can be obtained by adding two elements of the reduced set, i.e. the remaining multiples are generated dynamically [11]. Using secondary set of multiples reduces the cost of the generating and storing the set of multiples needed but affects the complexity of the design since more addition operations are needed to generate the partial product. Different secondary sets have been proposed. Even multiples $2A, 4A, 6A, 8A$ is an example of a secondary set where odd multiplies $3A, 5A, 7A, 9A$ are generated on demand from the corresponding even multiplies as

$mX+X, m = 2, 4, 6, 8$ [1]. Other examples of the reduced set are $2A, 3A, 4A, 8A$ [11] and $A, 2A, 4A, 8A$ [15]. Although the last reduced set is smaller and simpler in calculations (simple shifting to BCD8421 coding), it requires more than two addition operations for some multiples (requires addition of three multiples $A, 2A$, and $4A$ to generate $7A$) which increase the delay and area needed in this design. A reduced set of $A, 2A, 4A, 5A$ can be generated without carry propagation over the whole number (the carry will propagate to the next significant only) [11]. In order to avoid the complexity of multiples generation, the multiplier can be recoded to a signed-digit (SD) representation. In [16], the multiplier digits are recoded as $y_i = yH_i + yL_i$ with $yH_i \in \{0, 5, 10\}$ and $yL_i \in \{-2, -1, 0, 1, 2\}$. Generation of the reduced set $\{A, 2A, 5A, 10A\}$ for this design requires few levels of combinational logic without carry propagation. It also requires an additional 10's complement operation to generate negative multiples. The recurrence equation for algorithms that use reduced sets is :

$$P_{i+1} = (P_i + r^n(A \times b'_i + A \times b''_i)) \times 10^{-1}$$

where $b_i = b'_i + b''_i$ is the i^{th} digit of the multiplier starting from the least significant digit and Ab'_i, Ab''_i are selected from the secondary set of multiples, $P_0 = 0$, $0 \leq i \leq n$, and n is the number of digits. Another approach for generation of partial products is to perform BCD digit-by-digit multiplication of input operands using lookup tables [12] or combinational logic [13]. A recoding of the operands digits by

Table 2.1: BCD Codings

	BCD-8421	BCD-5421	BCD-4221 (S1)	BCD-4221 (S2)	BCD-5211 (S1)	BCD-5211 (S2)
0	0000	0000	0000	0000	0000	0000
1	0001	0001	0001	0001	0001	0010
2	0010	0010	0100	0010	0100	0100
3	0011	0011	0101	0011	0101	0101
4	0100	0100	0110	1000	0111	0111
5	0101	1000	1001	1001	1000	1000
6	0110	1001	1010	1010	1010	1001
7	0111	1010	1011	1011	1011	1100
8	1000	1011	1110	1110	1110	1101
9	1001	1100	1111	1111	1111	1111

a signed-digit radix-10 is suggested to reduce the magnitude range [14]. Recoding of both operands speeds up and simplifies the generation of signed-digit partial products using simplified lookup tables and combinational logic. This approach is more suited for serial multiplier implementations because of its high hardware demand that makes them impractical for parallel designs [2].

Another approach to generate multiples is to recode the multiplicand digits. Recoding simplifies the generation of multiples and negative multiples. Different BCD coding schemes are shown in Table 2.1. An advantage of using BCD-4221 and BCD-5211 is that they are self-complementing (i.e. their 9's complement is simply obtained by bit inversion which simplifies the generation of negative multiples) [1]. The generation of $2X$ using BCD-5421 - unique encoding shown in Table 2.1 - is simply done by a 1-bit left shift with the result coded in BCD 8421 [1]. For example, shifting the code (0100), that represents number 4 in BCD-5421 according to Table

2.1, will result in (1000) which is equivalent to number 8 in BCD-8421. Similarly, number 7 (1010) will produce two digits, (0001) and (0100) which represent number 14 in BCD-8421. Likewise, the generation of $5X$ using BCD-4221 (both S1 and S2) is simply by 3-bit left shift with the result coded in BCD 5211 [1]. Figure 2.1 shows an example of this operation. Another way to generate $2X$ is to encode digits in BCD-5211 (S2) and shift to the left by one bit with the result encoded using BCD-4221 (S2) [1]. For example, shifting the code (0101), that represents number 3 in BCD-5211(S2) according to Table 2.1, will result in (1010) which is equivalent to number 6 in BCD-4221(S2). The recoding could be done through simple combinational logic [1] [2]. This recoding of multiplicand digits can be used with a signed-digit (SD) recoding of the multiplier digits to improve the performance of the multiplier. In [1] [2], the recoding of multiplier in SD radix-4, SD radix-5 and SD radix-10 is proposed.

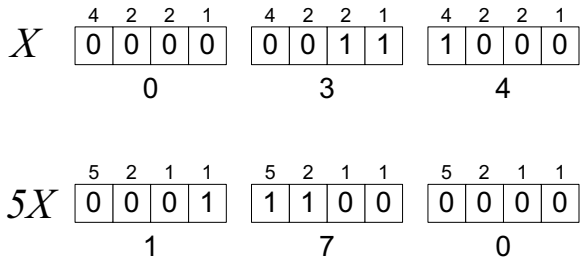


Figure 2.1: Example of calculation of $\times 5$ for decimal operands coded in BCD 4221

2.2.2 Reduction of Partial Products

In this stage, appropriate set of multiples generated in the first stage are selected and added together with the partial products accumulated during previous stages. The type of adder used in this stage depends on the encoding of the decimal digits. The reduction stage is the main stage that affects the time required to generate the result of the multiplier since it should be executed in every cycle. Designers try to optimize adders to enhance the delay of multipliers. Usually the output of the reduction stage is represented in redundant form in order to reduce the time required to calculate the complete result. Decimal addition algorithms and procedures are described in section 2.3. These algorithms include unsigned BCD representations and signed-digit representations.

2.2.3 Final Carry Propagate Addition

The reduction stage sometimes produces the final result in redundant form (in sum S and carry C component) at the end. This stage converts the final result into a non-redundant form. The sum and carry components are passed to a carry propagate adder that perform the addition operation according to addition algorithms described in section 2.3 (speculative decimal addition algorithm and direct decimal addition algorithm). [4] [1]

2.3 Decimal Addition

2.3.1 Unsigned BCD Addition

In BCD-8421 encoding, ten out of sixteen possible combinations are used (i.e. six combination from '1010' to '1111' are unused). Because of this, decimal addition is more complicated than binary addition. The basic decimal addition algorithm is described in Figure 2.2. If the sum of two BCD digits exceeds the value of nine and gets into the unused range, the sum should be wrapped around to the appropriate value and the carry out bit should be set [11]. This problem can be resolved by additional hardware to detect when the sum digit exceeds the allowed range or by adding six to the final sum and check if a carry-out is generated or not. In this subsection, the operands are considered as unsigned numbers(i.e. addition of negative values and subtraction is not considered). Another alternative way to skip the unused combinations is by biasing one BCD digit by six, perform the addition operation with the second BCD digit and then make a correction operation (subtracting the +6 bias) if the carry-out C_{i+1} is zero. This algorithm is called *speculative decimal addition* detailed in figure 2.3. The addition of six in each digit in one of the operands allows the use of binary adders, that are already available, for each decimal digit. The operation of adding six (biasing) can be performed digitwise using a single level of combinational logic according to the following Boolean equations:

Algorithm: Basic Decimal Addition (S=X+Y)

Input: $X = \sum_{i=0}^{d-1} X_i.10^i, Y = \sum_{i=0}^{d-1} Y_i.10^i, C_0 = C_{in}$

For (i=0; i<d; i++){

$C_{i+1} = \lfloor (X_i + Y_i + C_i)/10 \rfloor$

$S_i = \mathrm{mod}_{10}(X_i + Y_i + C_i)$

}

Figure 2.2: Basic Addition Algorithm

Algorithm: Speculative Decimal Addition (S=X+Y)

Input: $X = \sum_{i=0}^{d-1} X_i.10^i, Y = \sum_{i=0}^{d-1} Y_i.10^i, C_0 = C_{in}$

For (i=0; i<d; i++){

$S_i^* = \mathrm{mod}_{16}(X_i + Y_i + 6 + C_i)$

$C_{i+1} = \lfloor (X_i + Y_i + C_i)/10 \rfloor = \lfloor (X_i + Y_i + 6 + C_i)/16 \rfloor = \lfloor S_i^*/16 \rfloor$

$S_i = \mathrm{mod}_{10}(X_i + Y_i + C_i) = \begin{cases} \mathrm{mod}_{16}(S_i^* - 6) & , \text{ if } C_{i+1} = 0 \\ S_i^* & , \text{ otherwise} \end{cases}$

}

Figure 2.3: Speculative Decimal Addition Algorithm

$$X_i + 6 = \{(x_{i,3} + x_{i,2} + x_{i,1}), (\overline{x_{i,2} \oplus x_{i,1}}), (\overline{x_{i,1}}), (x_{i,0})\}$$

where X_i is a 4-bit decimal digit ($X_i = x_{i,3}\ x_{i,2}\ x_{i,1}\ x_{i,0}$) and $X_i \in [0, 9] \Rightarrow X_i + 6 \in [6, 15]$ is represented in BCD excess-6 [1]. Instead of biasing one of the operands, both operands can be represented in BCD excess-3 format which will lead to the same result. Likewise, the correction stage (subtracting six from each digit) can be implemented using a single level of combinational logic as follows:

$$S_i - 6 = \{(s_{i,3}.s_{i,2}.s_{i,1}), (s_{i,2} \oplus s_{i,1}), (\overline{s_{i,1}}), (s_{i,0})\}$$

where S_i is a 4-bit decimal digit ($S_i = s_{i,3}\ s_{i,2}\ s_{i,1}\ s_{i,0}$) and $S_i - 6 \in [0, 9]$ Although this algorithm allows the use of binary adders, the main drawback of this algorithm is the propagation delay in the critical path which includes the delay of operands setup, the carry computation, the sum and the post-correction stage [1]. This delay can be reduced by parallelizing evaluation and correction of BCD sum digits with the carry computation using a hybrid prefix tree/carry-select adder which is implemented in the fixed-point unit of IBM z900 and IBM z990 microprocessors [10][18].

Another algorithm of decimal addition is *direct decimal addition* that accepts two 4-bit BCD digits X_i and Y_i as inputs in BCD-8421 along with $C_i[0]$ one bit carry-in and directly produce the a 4-bit sum digit S_i in BCD-8421 and 1-bit carry-out $C_{i+1}[0]$ [11]. In direct decimal addition algorithm, the decimal carry is obtained

directly through an implementation of a decimal carry recurrence ($C_{i+1} = G_i + A_i C_i$) which depends on the decimal carry generate G_i and decimal carry alive A_i signals that represent the conditions for generating and propagating of a decimal carry [1]. This algorithm does not allow the use of binary adders. Figure 2.4 describes the direct decimal algorithm.

The generate G_i and alive signals A_i are given by:

$$G_i = G_i^U + A_i^U \,.\, x_{i,0} \,.\, y_{i,0} \tag{2.1}$$

$$A_i = A_i^U .(x_{i,0} + y_{i,0}) \tag{2.2}$$

where the upper decimal carry-generate G_i^U and upper decimal carry-alive A_i^U signals are obtained from the three most significant bits of the decimal digit as follows:

$$G_i^U = x_{i,3}(y_{i,3} + y_{i,2} + y_{i,1}) + y_{i,3}(x_{i,2} + x_{i,1}) + x_{i,2}y_{i,2}(x_{i,1} + y_{i,1}) \tag{2.3}$$

$$A_i^U = x_{i,3} + y_{i,3} + x_{i,2}y_{i,2} + (x_{i,3} + y_{i,3})x_{i,1}y_{i,1} \tag{2.4}$$

Also the upper decimal carry-generate G_i^U and upper decimal carry-alive A_i^U signals can be expressed in term of the binary carry-generate ($g_{i,j} = x_{i,j}.y_{i,j}$) and

binary carry-alive ($a_{i,j} = x_{i,j} + y_{i,j}$) signals as follows:

$$G_i^U = g_{i,3} + g_{i,2}.a_{i,1} + a_{i,3}(a_{i,2} + a_{i,1}) \tag{2.5}$$

$$A_i^U = a_{i,3} + g_{i,2} + a_{i,2}.g_{i,2} \tag{2.6}$$

The sum digit in direct decimal addition S_i given by $S_i = mod_{16}(X_i + Y_i + C_i + 6.C_{i+1})$ can be evaluated directly from X_i and Y_i and the computed carries using combinational logic [1]. The bits of the sum digit S_i are given as follow:

$$c_{i,1} = g_{i,0} + a_{i,0}.C_i \tag{2.7}$$

$$C_{i+1} = G_i^U + A_i^U.c_{i,1} \tag{2.8}$$

$$g_{i,j} = x_{i,j}.y_{i,j} \tag{2.9}$$

$$a_{i,j} = x_{i,j} + y_{i,j} \tag{2.10}$$

$$p_{i,j} = x_{i,j} \oplus y_{i,j} \tag{2.11}$$

$$S_i = \begin{cases} s_{i,3} = (g_{i,3} + p_{i,2}.p_{i,1})\overline{p_{i,3}}.c_{i,1} + A_i^U.\overline{G_i^U}\,\overline{c_{i,1}} \\ s_{i,2} = (p_{i,2} \oplus p_{i,1}.\overline{A_i^U}).c_{i,1} + (p_{i,2} \oplus \overline{p_{i,1}}.G_i^U).\overline{c_{i,1}} \\ s_{i,1} = \overline{p_{i,1} \oplus A_i^U}.c_{i,1} + (p_{i,1} \oplus G_i^U).\overline{c_{i,1}} \\ s_{i,0} = p_{i,0} \oplus C_i \end{cases}$$

Another way to add decimal numbers is to use recoding of decimal digits into

redundant non-conventional encodings (examples are shown in table 2.1). For Example, input operands are encoded in BCD-4221 or BCD-5211 and fed to a binary carry-save adder to produce a sum component S and a carry component H as:

$$X_i + Y_i + C_i = \sum_{j=0}^{3}(s_{i,j} + 2h_{i,j})r_j = \sum_{j=0}^{3} s_{i,j}r_j + 2\sum_{j=0}^{3} h_{i,j}r_j = S_i + 2H_i$$

where (r_3, r_2, r_1, r_0) is either $(4, 2, 2, 1)$ or $(5, 2, 1, 1)$ and

$$s_{i,j} = x_{i,j} \oplus y_{i,j} \oplus c_{i,j} \tag{2.12}$$

$$h_{i,j} = x_{i,j}.y_{i,j} + (x_{i,j} + y_{i,j}).c_{i,j} \tag{2.13}$$

$H_i \in [0, 9], S_i \in [0, 9]$ are the decimal carry and sum digits at position i respectively [2][1]. Sum and carry components do not need any correction (i.e. further addition or subtraction operations). There are three different implementations of addition operation of recoded digits using binary carry-save adders described below:

Input and output operands encoded in BCD-4221 : In this case, a one-decimal-digit carry-save adder consists of 4-bit binary carry-save adder and a digit recoder from (4221) to (5211). The decimal digit CSA receives three operands encoded in BCD-4221 and produce two operands in the same encoding. However, the carry component H needs to be multiplied by 2 before further computations. This can be done by recoding the carry word into BCD-5211 then

shift it to the left by one bit. [2]. Figure 2.5 shows this case. For example, addition operation of (1110) = 8, (1100) = 6, and (1001) = 5 is shown in Table 2.2

Table 2.2: Example of Decimal Addition with recoding: Input and Output operands encoded in BCD-4221 (case 1)

	(4221)	(4221)
	0	8
Input(X_i)	0000	1110
	0	6
Input(Y_i)	0000	1100
	0	5
Input(C_i)	0000	1001
	0	7
Sum(S_i)	0000	1011
	0	6
Carry($\grave{H}_i$)	0000	1100
Carry recoding		
	(5211)	(5211)
	0	6
Carry(H_i)	0000	1010
Carry shifting		
	(5211)	(5211)
	1	2
Carry(H_i)	0001	0100

Input and output operands encoded in BCD-5211 : This case is much similar to the previous one. Its decimal digit CSA consists of 4-bit binary carry-save adder and a digit recoder from (5211) to (4221). The carry component H needs to be multiplied by 2 before further computations (i.e. to get $2H$). This is achieved by shifting the carry component by one bit to the left followed by recoding from BCD-4221 to BCD-5211 [1]. Figure 2.6 shows this case. An

Algorithm: Direct Decimal Addition (S=X+Y)

Input: $X = \sum_{i=0}^{d-1} X_i.10^i, Y = \sum_{i=0}^{d-1} Y_i.10^i, C_0 = C_{in}$

For (i=0; i<d; i++) {

$$G_i = \begin{cases} 1, & \text{if } (X_i + Y_i > 9) \\ 0, & \text{Otherwise} \end{cases}$$

$$A_i = \begin{cases} 1, & \text{if } (X_i + Y_i \geq 9) \\ 0, & \text{otherwise} \end{cases}$$

$$C_{i+1} = \lfloor (X_i + Y_i + C_i)/10 \rfloor = G_i + A_i C_i$$

$$S_i = \mathrm{mod}_{10}(X_i + Y_i + C_i) = \mathrm{mod}_{16}(X_i + Y_i + C_i + 6.C_{i+1})$$

}

Figure 2.4: Direct Decimal Addition Algorithm

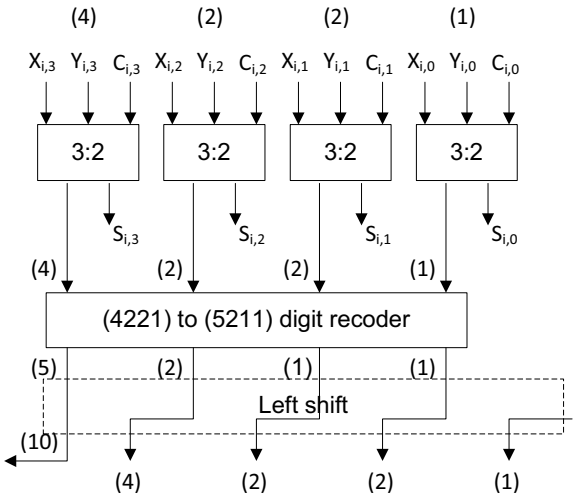


Figure 2.5: Decimal Digit (4-bit) 3:2 CSA: Input and Output operands encoded in BCD-4221 (case 1)

example for addition operation of (1110) = 8, (1100) = 7, and (0110) = 3 is shown in Table 2.3.

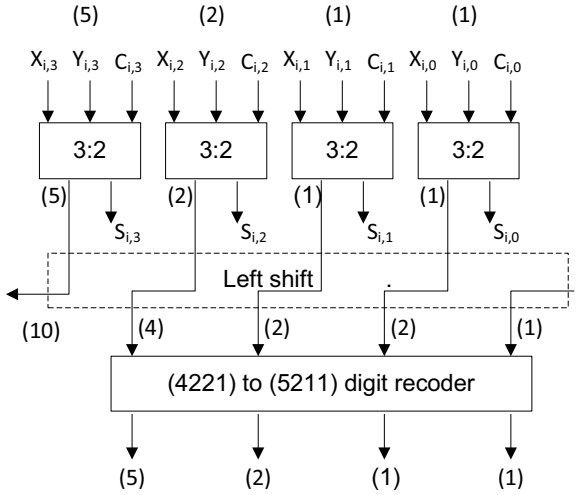


Figure 2.6: Decimal Digit (4-bit) 3:2 CSA: Input and Output operands encoded in BCD-5211 (case 2)

Mixed input and output encodig : In this case, the decimal digit CSA needs only 4-bit binary carry-save adder. The input operands are encoded in BCD-5211 and the output operands (sum S and carry H components) are encoded in BCD-5211. The carry component is shifted to the left by one bit postion to get $2H$ encoded in BCD-4221 [1]. Figure 2.7 shows this case. The example is shown in Table 2.3 without the recoding phase.

The recoding process can be achieved through a simple gate level. The logical

Table 2.3: Example of Decimal Addition with recoding: Input and Output operands encoded in BCD-5211 (case 2)

	(5211)	(5211)
Input(X_i)	0 0000	7 1100
Input(Y_i)	0 0000	7 1100
Input(C_i)	0 0000	3 0110
Sum(S_i)	0 0000	3 0110
Carry($\grave{H}_i$)	0 0000	7 1100
Carry shifting		
	(4221)	(4221)
Carry(H_i)	1 00001	4 1000
Carry recoding		
	(5211)	(5211)
Carry(H_i)	1 0001	4 0111

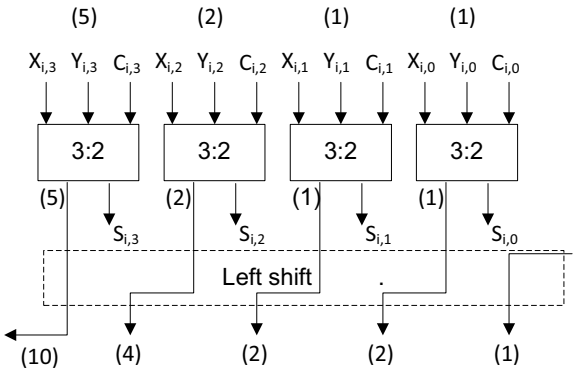


Figure 2.7: Decimal Digit (4-bit) 3:2 CSA: Mixed Input and Output operands encoding (case 3)

expressions for BCD-4221 to BCD-5211 recoding is given by:

$$w_{i,3} = h_{i,3}.(h_{i,2} + h_{i,1} + h_{i,0}) + h_{i,2}.h_{i,1}.h_{i,0} \quad (2.14)$$

$$w_{i,2} = h_{i,2}.h_{i,1}.\overline{h_{i,3} \oplus h_{i,0}} + (h_{i,3}.\overline{h_{i,0}}) \oplus h_{i,2} \oplus h_{i,1} \quad (2.15)$$

$$w_{i,1} = h_{i,2}.h_{i,1}.\overline{h_{i,3} \oplus h_{i,0}} + h_{i,3}.\overline{h_{i,0}}.\overline{h_{i,2} \oplus h_{i,1}} \quad (2.16)$$

$$w_{i,0} = (h_{i,2}.h_{i,1}). \oplus h_{i,3 \oplus h_{i,0}} \quad (2.17)$$

where the input $H = (h_{i,3}, h_{i,2}, h_{i,1}, h_{i,0})$ is encoded in BCD-4221 and the output $W = (w_{i,3}, w_{i,2}, w_{i,1}, w_{i,0})$ is encoded in BCD-5211 [2]. The reverse digit recoding from BCD-5211 to BCD-4221 is easily implemented using a single full adder where the most significant bit $h_{i,3}$ and the two least significant bits $h_{i,1}, h_{i,0}$ are added according to the following equation:

$$H_i(5211) = h_{i,3}(4+1) + h_{i,2}(2) + h_{i,1}(1) + h_{i,0}(1) \quad (2.18)$$

$$= h_{i,3}(4) + h_{i,2}(2) + (h_{i,3}(1) + h_{i,1}(1) + h_{i,0}(1)) \quad (2.19)$$

with $w_{i,3} = h_{i,3}$, $w_{i,2} = h_{i,2}$ and $(h_{i,3}(1) + h_{i,1}(1) + h_{i,0}(1)) = 2 \times w_{i,1} + w_{i,0}$ where the input $H = (h_{i,3}, h_{i,2}, h_{i,1}, h_{i,0})$ is encoded in BCD-5211 and the output $W = (w_{i,3}, w_{i,2}, w_{i,1}, w_{i,0})$ is encoded in BCD-4221 [1].

2.3.2 Signed Digit Decimal Addition

The second category of decimal addition algorithms is *signed-digit addition*. There are many algorithms and hardware implementations for signed-digit addition based on the codes used for the decimal digits. Signed-digit representations of decimal numbers allow redundant representations of some decimal numbers. Signed-digit addition does not have a carry propagation, but it has a transfer quantity that is independent from the input carry and only propagate to the next digit position. In [26], Antonin Svoboda introduced a signed-digit adder that uses a symmetric digit set [-6,+6]. Svoboda suggested 5-bits representations for a svoboda-encoded digit $x_i \in [-6, 6]$ given by:

$$X_i = 3x_i (mod\ 31) \quad for\ 0 \leq X_i \leq 31$$

so that

$$X_i = 3x_i \quad for\ x_i \geq 0$$

and

$$X_i = 31 - 3x_i \quad for\ x_i \leq 0$$

Binary codes for decimal digits based on Svoboda rule are shown in Table 2.4.

These Svoboda proposed codes have the following important properties:

Table 2.4: Svoboda Signed-digit Codes

x_i	Binary Code					X_i
	e	d	c	b	a	
+6	1	0	0	1	0	+18
+5	0	1	1	1	1	+15
+4	0	1	1	0	0	+12
+3	0	1	0	0	1	+9
+2	0	0	1	1	0	+6
+1	0	0	0	1	1	+3
+0	0	0	0	0	0	0
-0	1	1	1	1	1	31
-1	1	1	1	0	0	28
-2	1	1	0	0	1	25
-3	1	0	1	1	0	22
-4	1	0	0	1	1	19
-5	1	0	0	0	0	16
-6	0	1	1	0	1	13

1. Sign inversion is done by complementing bits.

2. Positive digits have even parity while negative digits have odd parity. Moreover, the MSB can be used as a sign bit (except for +6 and -6).

3. The marginal digits (+6,-6) are distinguished by the relations:

$$(x_i = +6) \Leftrightarrow (\overline{a}b\overline{c} = 1)$$

$$(x_i = -6) \Leftrightarrow (\overline{a} + b + \overline{c} = 1)$$

$$(x_i = \pm 6) \Rightarrow (a + b + c + d \equiv 1 (mod\ 2))$$

4. When (+6 = 10 - 4) is fed as an input to the adder, a transfer value of +10

from order i to order $i+1$ is generated as a single additive unit and the value of current digit (+6) should be replaced by (-4). Closer investigation of the codes of both values (+6 and -4) shows that this conversion is simply achieved by complementing the least significant bit (i.e. change a from 0 to 1).

5. Likewise when (-6 = -10 + 4) is fed as an input to the adder, a transfer value of -10 from order i to order $i+1$ is generated as a single additive unit and the value of current digit (-6) should be replaced by (+4). Closer investigation of the codes of both values (-6 and +4) shows that this conversion is simply achieved by complementing the least significant bit (i.e. change a from 1 to 0).

The hardware structure of this adder is shown in Figure 2.8. This hardware structure is meant for description purposes and not for synthesis as indicated by the author. As shown in the hardware structure, this algorithm uses binary adders to add the input digits and to add the transfer digit but it uses unconventional encoding of decimal digits so a recoding operation is needed. Also, it uses 5-bits to represents digits set [-6,+6] which indicates inefficiency in representation (i.e. 13 combinations are used out of 32 possible combinations).

A Second algorithm was proposed in 2007 by John Moskal, Erdal Oruklu and Jafar Saniie [27]. In this algorithm, decimal signed-digit number X_i is represented by two 4-bits quantities (X_i^+, X_i^-) where X_i^+ represents the positive component

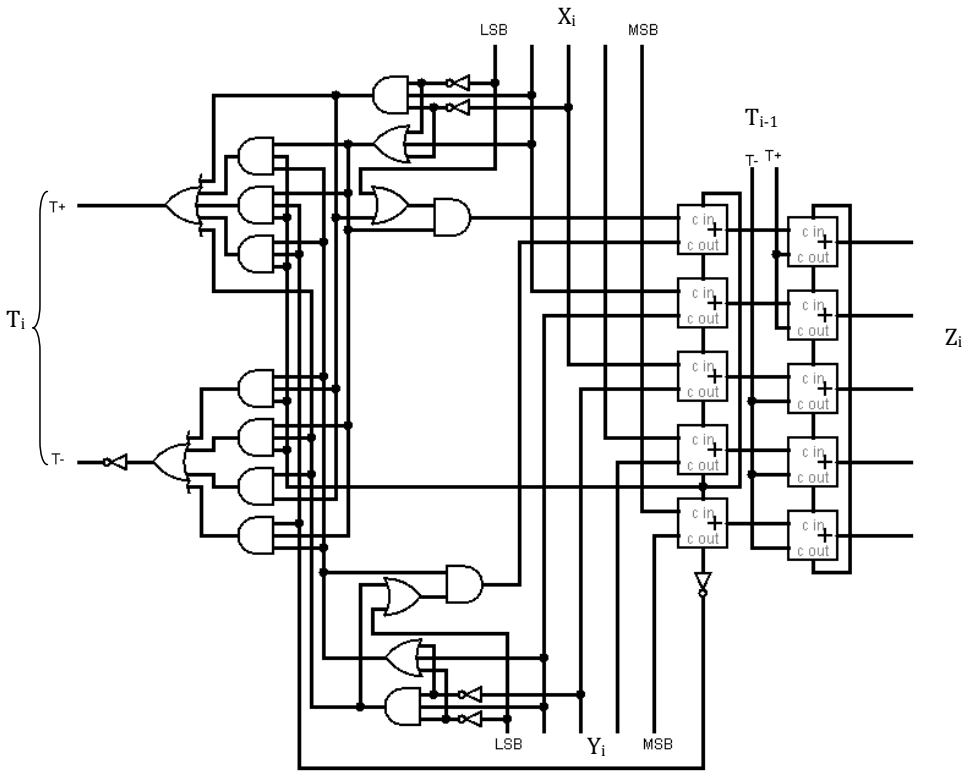


Figure 2.8: Svoboda Signed-digit Decimal Adder (Descriptive Diagram, i.e. not for synthesis purposes)

and X_i^- represents the negative component. Both components are encoded using BCD encoding. To obtain the numerical value of the number, subtract the negative component from the positive component (i.e. $X_i = X_i^+ - X_i^-$). The digit set of this algorithm is from +9 to -9 inclusive. To add two decimal digits, if the value of interim sum exceeds (1) then a transfer digit is set and a correction factor of 10 is subtracted. Similar operation is applied for negative quantities. When the interim sum is less than (-1), then the transfer digit is set to (-1) and a correction factor of (+10) is added. The transfer digit is set to zero in remaining cases when the interim sum is either (0), (+1) or (-1). Thus, the transfer digit depends on the input values only regardless of the transfer digit from the previous digit which means there is no carry propagation. Since the interim sum is in the range of [+18,-18], this guarantees that the output digit will remain in the same range of the inputs [+9,-9] (after adding the transfer digit performing the correction operation).

$$U_i = X_i + Y_i$$

$$t_i = \begin{cases} 1 & if\ (X_i + Y_i) > 1 \\ 0 & if\ |X_i + Y_i| \leq 1 \\ \overline{1} & if\ (X_i + Y_i) < -1 \end{cases}$$

$$S_i = U_i + t_{i-1} - 10 \times t_i$$

The architecture of the adder depicting this algorithm is shown in Figure 2.9.

The design consists of three functional blocks: compressor block, selector block and correction block. In the compressor block, the initial addition operation between two operands is performed and the interim sum is produced. Selector block chooses one of two intermediate values generated by the first block to perform further calculations and compute the transfer digit. Correction block receives the transfer digit from less order digit, applies it to the interim sum, adjusts the result of the transfer-out digit and reduces the final result to 4 bits. These functional blocks use the following components:

Combined Compressor: is a single bit compressor that takes two SD values and represents their sum in two equivalent forms. There are two types of compressors: Type "'A"' compressor represents $X_i + Y_i$ as sum of C_{i+1}^-, C_{i+1}^+ and S_i^- and type "'B"' compressor represents $X_i + Y_i$ as sum of C_{i+1}^-, C_{i+1}^+ and S_i^+.

Sign Detector: used to detect the sign of the intermediate result and determine the transfer digits.

SD Binary Adder: takes two inputs, one SD number and one binary number, and provides positive carry-out C_{i+1}^+ and negative sum S_i^-.

Reduction Circuit: used to correct the result coming from the binary adder and reduce it from 5 bits into 4 bits.

The inefficient representation of decimal digits is considered as a disadvantage (19 values are represented by 8 bits). Although this increases the redundancy, it

increases the area of components involved in computation. In the hardware design, some components generates values that are not used which means that the hardware is not fully utilized.

Another signed-digit decimal addition algorithm was proposed by Behrooz Shirazi, David Y.Y. Yun, and Chang N. Zhang in 1989 [28]. This algorithm uses 4 bits to represent digits in the digit set D from -7 to +7 coded in BCD. Negative values are represented in 2's complement. Table 2.5 shows digits and their corresponding redundant BCD (RBCD) codes.

Table 2.5: Redundant BCD (RBCD) Signed-digit Codes

Digit	RBCD code	Digit	RBCD code
0	0000		
1	0001	-1	1111
2	0010	-2	1110
3	0011	-3	1101
4	0100	-4	1100
5	0101	-5	1011
6	0110	-6	1010
7	0111	-7	1001

Addition operation suggested by this algorithm consists of three stages. In the first stage, operands are fed to a binary carry-save adder to generate interim sum which should be in the subset $\grave{D} = D - \{7, -7\} = \{6, 5, ..., 0, ..., -5, -6\}$ and transfer digit $T \in \{-1, 0, 1\}$. Then, the interim sum is exposed to a correction operation if the output digit goes beyond the boundaries of the digit set. When the interim sum digit is within the ranges $\{8, 9, ..., 14\}$, then addition of (-6) and setting the transfer carry to (+1) are performed in correction stage. On the other hand, when

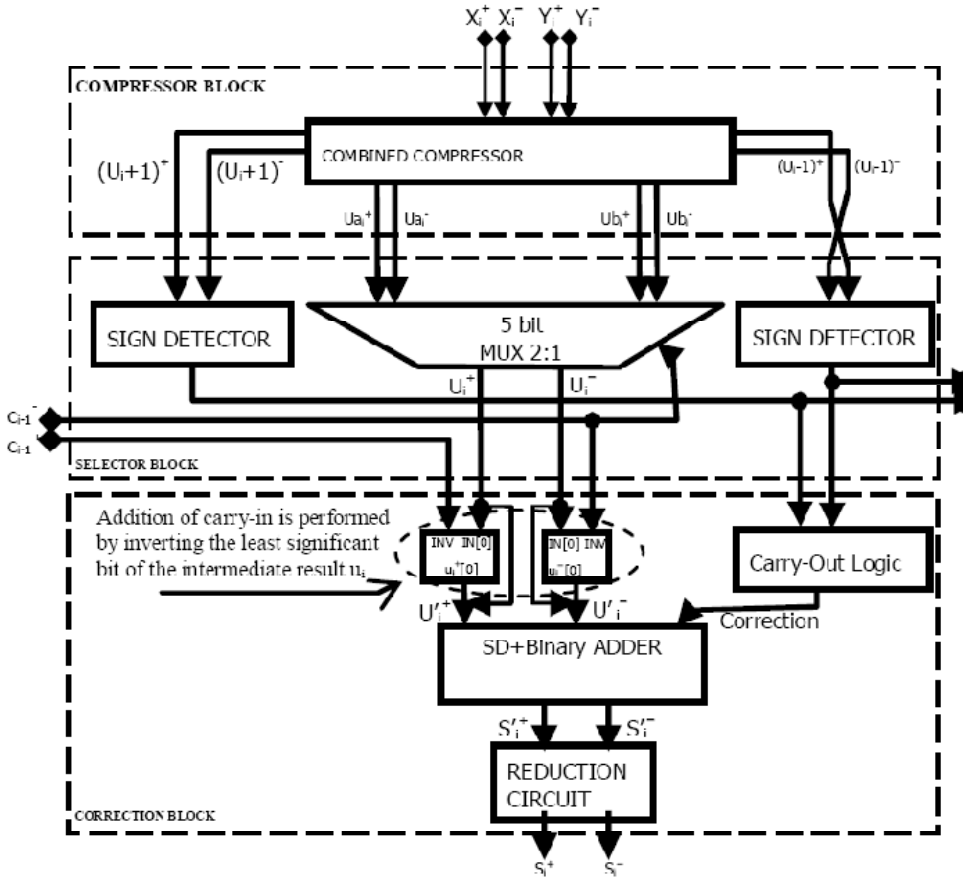


Figure 2.9: Decimal Adder Architecture (Moskal Algorithm)

the interim sum digit enters the range $\{-8, -9, ..., -14\}$, then addition of (+6) and setting the transfer carry to (-1) are performed in correction stage. By combining these three cases (i.e. intermediate sum $\in \{6, 5, ..., 0, ..., -5, -6\}$, intermediate sum $\in \{7, 8, ..., 14\}$, and intermediate sum $\in \{-7, -8, ..., -14\}$), for any pair of RBCD digits added, there are 9 possible correction factors used to correct the intermediate sum, namely $\{-7, -6, -5, -1, 0, 1, 5, 6, 7\}$.

The large set of correction factors and the condition on them makes the decision more complicated. Since 2'complement representation is common and allows the use of binary adders, this addition algorithm is more suitable to be integrated with binary adders.

An enhanced algorithm of the previous one was proposed by Jeff Rebacz, Erdal Oruklu, and Jafar Saniie in [29]. The main enhancements lie in digit set, carry detection unit and correction stage. The new digit set is [9,-9] inclusive represented in 2'complement 5 bits vector. The operands are added by using Carry-look-ahead adder to generated 6-bits interim sum u_i. Then, two level of logic is used to determine the transfer carry t_i (+1 when $u_i > 7$, -1 when $u_i < -8$, 0 otherwise). After that, the correction vector ($=-10 \times t_i + t_{i-1}$) is computed and added. The Hardware proposed for this adder is shown in Figure 2.10.

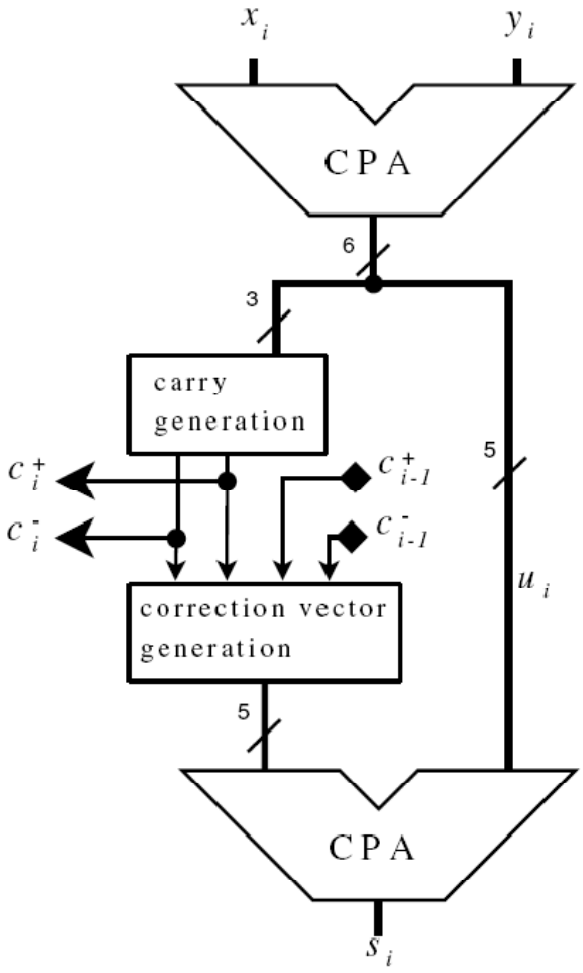


Figure 2.10: Enhanced RBCD Decimal Adder Architecture

2.4 IEEE 754-2008 Standard for Floating-Point

The IEEE 754-2008 standard is a revision of the IEEE 754-1985 standard for floating-point operations in computer systems [9]. The standard specifies formats, methods, standard and extended functions for floating-point arithmetic in computer systems with single, double, extended, and extendable precision. Also the standard includes recommended formats for computation and data interchange between binary and decimal representation. The standard could be implemented in hardware, software, or a combination of both. Exception conditions and their handling are included [9].

IEEE 754-2008 adds new operations that were not included in the old standard IEEE 754-1985 like fused multiply add, and adds new binary types (16-bit and 128-bits) in addition to decimal specifications [1]. In this overview, we summarize some of the specifications for decimal floating-point operations.

2.4.1 Decimal Specifications in IEEE 754-2008 Standard

Both decimal formats and encodings are an integral part of IEEE 754-2008 standard [1]. In decimal floating-point format, a number can have multiple representations which are called *cohort* [9]. Members of *cohort* are distinct representation of the same floating-point value.

The standard defines interchange formats as formats that have a specific fixed-width encoding defined in the standard, and classifies them as follows:

Basic Formats which are the formats available for arithmetic operations [1]. Two basic formats are defined for decimal operations of length 64 bits (decimal64) and 128 bits (decimal128) [9].

Storage Formats which are smaller than basic format and not required for arithmetic. The standard defines only one storage format with length of 32 bits (decimal32) [1].

Extended Precision Formats which have both wider range and wider precision to extend a supported basic format [9].

Also the standard mentioned extendable formats that are under user control. Users can specify the range and precision of that format according to their design needs [9]. To interchange data represented in an extendable format, it needs to be converted into one of the interchange formats [1].

Decimal Floating-point (DFP) representation has two forms: scientific and financial. In the scientific form, a DFP is represented by a triplet s, e, M corresponding to a value of:

$$(-1)^s \times M \times 10^e$$

where $s \in \{0, 1\}$ is the sign of the number, $e_{min} \leq e \leq e_{max}$ is the integer exponent and $M < 10$ is the unsiged not normalized fractional part, in the form $M_0.M_1M_2M_3M_4...M_{p-1}$, $M_i \in \{0, 1, 2, ..., 9\}$ [1]. In the financial form, a DFP is

represented by a triplet s, q, C corresponding to a value of:

$$(-1)^s \times C \times 10^q$$

where $s \in \{0, 1\}$ is the sign of the number, the exponent (or quantum q) such that $e_{min} \leq q + p - 1 \leq e_{max}$ is the integer exponent and $C < 10^p$ is a string of decimal digits in the form $C_0C_1C_2C_3C_4...C_{p-1}$, $C_i \in \{0, 1, 2, ..., 9\}$ [1]. This type of representation is called *scaled decimal* and is needed in financial applications since the trailing zeros may represent extra information that should be preserved such as currency (cents or hallals) [1]. For example, the value 672.80 is represented as 67280 with a quantum of 2 (or an exponent -2), that is, as 67280×10^{-2}.The scaled decimal encoding is not unique since different combinations of quantum and integer coefficient can represent the same value (coefficient 030 with quantum 1 and coefficient 003 with quantum 3 represents the value 300) [1]. These different representations forms a *Cohort*. In general, If C is a multiple of 10 and $q < e_{max}$ then $\{s, q, C\}$ and $\{s, q + 1, C/10\}$ are two different representations of the same DFP number [1][9].

Although they seem to be different, both forms of representation (financial and scientific) are equivalent since e and M in the scientific form are equal to $q + p - 1$ and $C \times 10^p$ respectively in the financial form [1].

The *encoding* is the mapping between DFP number and a string of bits that

represent it. The standard defines the general layout for interchange formats and allows the integer coefficient to be represented in either a compressed form (Densely Packed Decimal, DPD) or pure binary form (Binary Integer Decimal, BID) [1]. The DPD encoding allows the compression of 3 decimal digits into 10 bits [23]. The general layout of DFP number encoded in k bits consists of the following three fields:

Sign Field S : 1-bit field that encodes the sign of the number.

Combinational Field G : w+5 bits field that contains biased exponent (w+2 bits) represented in binary ($E = q + bias$) and the 4 most significant bits of decimal coefficient (the most significant digit). The most significant bits of the exponent can not be $(11)_b$ [9].

Coefficient Field T : 10J bits field that encodes 3J digits of the coefficient encoded in DPD or binary integer value between 0 and 2^{10t-1}. This field with the most significant digit in the combinational field forms the $3 \times t + 1$ decimal digits of the integer coefficient [9].

Table 2.6 shows the values of k, w, p, t, and $bias$ for different interchange formats.

The combinational field G conveys the type of information stored as follows:

1. If the most significant bits of the combinational field G are "'11111"' (i.e. bits from G_0 through G_4 are ones), then the value is NaN (Not a Number).

Moreover, the next bit G_5 is used to distinguish between different types of NaN. When $G_5 = 1$, then the value is sNaN (signaling NaN), and when it is "'0"' then the value is qNaN (quit NaN). The remaining bits of G have no meaning and the T field carries the payload which is used to differentiate between various NaN. [9]

2. If the most significant bits of the combinational field G are "'11110"', then this represents the value of $(-1)^S \times (+\infty)$. The rest of G and T fields are negligible but it is preferred to be set to zeros [9].

3. For a finite number $r = (S, E - bias, C)$ that has a value $v = (-1)^S \times 10^{(E-bias)} \times C$, the value of C is combined from the leading significant digit or bits from combinational field G and coefficient field T and the biased exponent E is contained in combinational field G. The encoding of these information differs according to the type of the encoding used for the coefficient (binary or decimal). [9]

   (a) If the coefficient is encoded in *decimal*, then the bits from G_5 to G_{w+4} contain the least significant w bits of the biased exponent. The most significant two bits of the biased exponent E and the coefficient C are contained in the most significant bits of the combinational field G (i.e. bits from G_0 through G_4) and the coefficient field T as follows:

      i. If the bits $(G_0G_1G_2G_3G_4) = (1110x)$ or $(110xx)$, then the most sig-

nificant digit of the coefficient d_0 is $8 + G_4$ (i.e. $d_0 = 8$ or 9) and the leading biased exponent bits are $2G_2 + G_3$ (i.e. equal to 0,1, or 2) [9].

ii. If the bits $(G_0G_1G_2G_3G_4) = (0xxxx)$ or $(10xxx)$, then the most significant digit of the coefficient d_0 is $4G_2 + 2G_3 + G_4$ (i.e. d_0 in the range 0 to 7) and the leading biased exponent bits are $2G_0 + G_1$ (i.e. equal to 0,1, or 2) [9]. So, if T field is all zeros and $G = 00000, 01000, 10000$ then the value $= (-1)^S \times (+0)$ [9].

The remaining digits of the coefficient are encoded in T field by DPD

(b) If the coefficient is encoded in *binary*, then:

i. If ($G_0G_1 = 00$, 01, or 10), then bits from G_0 through G_{w+1} contain the biased exponent and significand is formed from bits G_{w+2} to the end of the encoding (including T field) [9].

ii. If ($G_0G_1 = 11$) and ($G_2G_3 = 00$, 01, or 10), then bits from G_2 through G_{w+3} contain the biased exponent and significand is formed from prefixing the 4 bits $(8 + G_{w+4})$ to T field [9].

The maximum value that can be represented in the integer coefficient by both encoding (decimal and binary) are the same, that is $10^{(3\times J+1)} - 1$ (or $10^{(3\times J)} - 1$ when T is used as payload in NaN). When the value exceeded the maximum, the value used for c is zero.[9]

2.4.2 Densely Packed Decimal (DPD)

Densely packed decimal encoding was introduced by Cowlishaw in 2002 [23]. It is a compact representation of decimal data that offers significant advantages over binary coded decimal encoding (BCD) which is used widely for encoding decimal data. DPD is an improvement to the encoding introduced by Chen and Ho in [24] in 1975 and called (Chen-Ho encoding) where three decimal digits at most are compressed into 10 bits providing 17% more space over BCD encoding [23]. Chen-Ho encoding was built based on Huffman code with leading bits to indicate which digit combination is used [23]. Chen-Ho encoding have some advantages over other encodings like: [23]

- conversion from BCD and vice versa is achieved by simple boolean operations (i.e. multiplication or division operations are not needed).
- fixed-length encoding which allows simpler encoding and decoding over variable length encodings like encoding proposed by Smith in [25].

Although Chen-Ho encoding works extremely fine when the length of digit is a multiple of 3, it is not suitable with other lengths because either digits are wasted or another encoding need to be used to represent the remaining digits of the number [23]. For example, consider a 256-bit register where 237 bits are available for the significand and 19 bits are used for sign and exponent. Only 230 bits can be

used to encode 69 digits by Chen-Ho encoding and the remaining 7 bits are wasted while they are sufficient to encode two more digits. DPD utilizes the advantages of Chen-Ho encoding and tries to avoid the drawbacks. It is also built using equivalent Huffman coding to Chen-Ho encoding scheme with better arrangement of bits that gives more advantages:

1. Compression of one or two decimal digits is allowed with 4 or 7 bits respectively. This means that any number of decimal digits could be encoded with DPD efficiently. For example, 71 decimal digits could be encoded in 237 bits (compared with 69 using Chen-Ho encoding).

2. Encodings of one or two decimal digits are right aligned with the 10 bits (remaining bits are all zeros). This allows simple expansion of a number by appending zeros to the left without recoding. Chen-Ho encoding scheme requires recoding to expand a one or two digits field into three digits.

3. The rearrangement of bits allows all single digits numbers (indeed all the numbers in the range from 0 through 79) to have the same code as in BCD. This simplifies the conversion process from and to BCD. In Chen-Ho encoding scheme, only numbers from 0 to 7 have this advantage.

Table 2.7 shows some decimal numbers with their BCD, Chen-Ho, and DPD codes.

Table 2.6: Decimal interchange format parameters

Parameter	decimal32	decimal64	decimal128	decimal$k$ ($k \geq 32$)
storage width in bits(k)	32	64	128	multiple of 32
precision in digits(p)	7	16	34	$9 \times k/32 - 2$
Maximum exponent(e_max)	96	384	6144	$3^{k/16+3}$
Encoding Parameters				
$bias, E - q$	101	398	6176	$e_{max} + p - 2$
sign bit	1	1	1	1
Combinational field in bits ($w + 5$)	11	13	17	$k/16 + 9$
Trailing significand field width in bits (t)	20	50	110	$15 \times k/16 - 10$
storage width in bits(k)	32	64	128	$1 + 5 + w + t$

Table 2.7: Comparision of Decimal Encoding

Decimal	BCD	Chen-Ho	Densely Packed
5	0000 0000 0101	000 000 0101	000 000 0101
9	0000 0000 1001	110 000 0000	000 000 1001
55	0000 0101 0101	000 010 1101	000 101 0101
99	0000 1001 1001	111 000 1001	000 101 1111
555	0101 0101 0101	010 110 1101	101 101 0101
999	1001 1001 1001	111 111 1001	001 111 1111

Table 2.8: Encoding 3 decimal digits to 10 densely packed decimal encoding

$d_{(1,0)}, d_{(2,0)}, d_{(3,0)}$	b_0, b_1, b_2	b_3, b_4, b_5	b_6	b_7, b_8, b_8
000	$d_{(1,1:3)}$	$d_{(2,1:3)}$	1	$d_{(3,1:3)}$
001	$d_{(1,1:3)}$	$d_{(2,1:3)}$	1	$0, 0, d_{(3,3)}$
010	$d_{(1,1:3)}$	$d_{(3,1:3)}, d_{(2,3)}$	1	$0, 1, d_{(3,3)}$
011	$d_{(1,1:3)}$	$1, 0, d_{(2,3)}$	1	$1, 1, d_{(3,3)}$
100	$d_{(3,1:2)}, d_{(1,3)}$	$d_{(2:1,3)}$	1	$1, 0, d_{(3,3)}$
101	$d_{(2,1:2)}, d_{(1,3)}$	$0, 1, d_{(2,3)}$	1	$1, 1, d_{(3,3)}$
110	$d_{(3,1:2)}, d_{(1,3)}$	$0, 0, d_{(2,3)}$	1	$1, 1, d_{(3,3)}$
111	$0, 0, d_{(2,1:2)}, d_{(1,3)}$	$0, 1, d_{(2,3)}$	1	$1, 1, d_{(3,3)}$

DPD encoding scheme accepts three decimal digits $d_1 d_2 d_3$ each encoded in BCD and having 4 bits and encode (compress) then into 10 bits. Table 2.8 shows how DPD expresses 3 decimal digits in 10 bits. In the table, the symbol $d_{1,0:3}$ refers to the bits from 0 through 3 in digit 1 where bit 0 is the most significant bit and bit 3 is the least significant bit and bits from b_0 to b_9 represent the code after compression.

The conversion operation described in Table 2.8 generates 1000 combinations, that represent values in the range from 0 through 999, out of 1024 possible combinations. Although, the rest of the bit patterns like 01x11x111x, 10x11x111x, or 11x11x111x are not generated, they are mapped to a value in the range of number from 0 to 999. [9]

Table 2.9 shows the decoding operation of a DPD code into its original decimal digits. The symbol "'x"' in the table denotes a "'don't care"' bit. Thus, all possible bits combinations are acceptable and mapped into 1000 possible 3-digit numbers with some redundancy. [9]

Table 2.9: Decoding 10-bits DPD Code into 3 Decimal Digits

b_6, b_7, b_8, b_3, b_4	d_1	d_2	d_3
0xxxx	$4b_0 + 2b_1 + b_2$	$4b_3 + 2b_4 + b_5$	$4b_7 + 2b_8 + b_9$
100xx	$4b_0 + 2b_1 + b_2$	$4b_3 + 2b_4 + b_5$	$8 + b_9$
101xx	$4b_0 + 2b_1 + b_2$	$8 + b_5$	$4b_7 + 2b_8 + b_9$
110xx	$8 + b_2$	$4b_3 + 2b_4 + b_5$	$4b_7 + 2b_8 + b_9$
11100	$8 + b_2$	$8 + b_5$	$4b_7 + 2b_8 + b_9$
11101	$8 + b_2$	$4b_3 + 2b_4 + b_5$	$8 + b_9$
11110	$4b_0 + 2b_1 + b_2$	$8 + b_5$	$8 + b_9$
11111	$8 + b_2$	$8 + b_5$	$8 + b_9$

2.5 Serial Arithmetic

Serial arithmetic refers to arithmetic operations where one digit of each input operand is received and one digit of the output is delivered at each clock cycle (Figure 2.11 shows the typical timing diagram for serial operations). Note that cycle 1 is the cycle in which the first digit of the output is delivered (by convention). The main advantage of using serial arithmetic is to reduce the number of interconnecting signal lines between modules in the design and to simplify their interfaces which influence the area and power dissipation. The time (number of clock cycles) required to receive the whole input digits and produce the output digit is the main drawback. Timing diagram in Figure 2.11 shows that the total execution time of serial operations consists of two components: the initial delay δ and the time to deliver all output digits n. *The initial delay* δ corresponds to the number of digits of input operands that are needed to determine the first result digit. Thus, the first output digit will be produced $\delta + 1$ cycles after the first digit of input operands is received. *The time to deliver all output digits* n is equal to the number of digits in the output as one digit is delivered each cycle. Consequently, the execution time is:

$$T_n = \delta + 1 + n$$

This delay can be reduced by computing the result as few digits of the operands are received instead of collecting all input digits before starting the operation which will include the delay of collection beside the operational delay.

There are two modes of serial arithmetic: Least-significant digit first (LSDF) mode and Most-significant digit first (MSDF). In LSDF, the digits of the input operands are applied from the least significant digit and the result is produced from the least significant digit as well. This mode is also known as right-to-left mode. On the other hand, in MSDF mode, the digit of input operands are received from the most significant position and the output digits are also produced from the most significant digit. This mode of serial arithmetic is known as *online arithmetic* and the corresponding delay is referred to as online delay. Figure 2.12 shows these two modes of serial arithmetic.

2.5.1 Online Arithmetic

Online arithmetic concept was introduced in 1977 by Ercegovac [20]. The name *on-line* comes from the possibility of overlapping between performing the operation with the digit-by-digit communication of operands/result [22]. This overlapping reduces the time required to perform a single operation and also reduces the time of long sequence of consecutive arithmetic operations by allowing operation overlapping.

For certain applications, online Arithmetic algorithms are favored because (i) it reduces the on-chip and off-chip connections (ii) it allows parallelism between

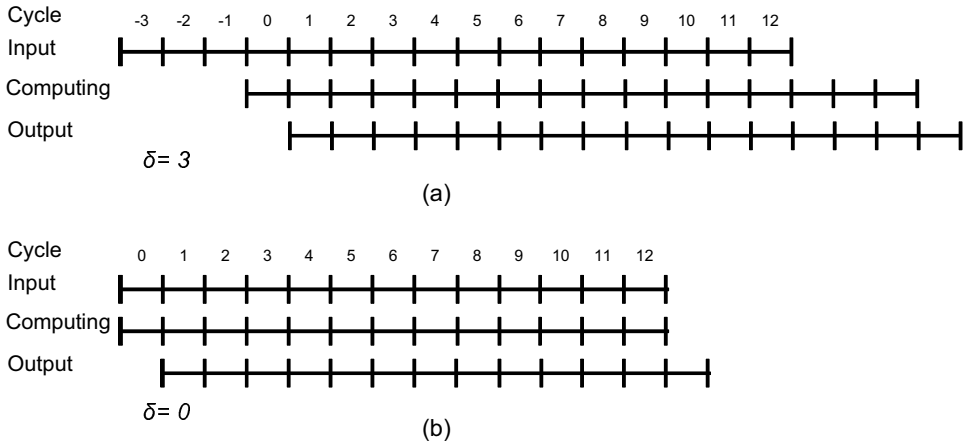


Figure 2.11: Timing characteristics of serial operations with n=12

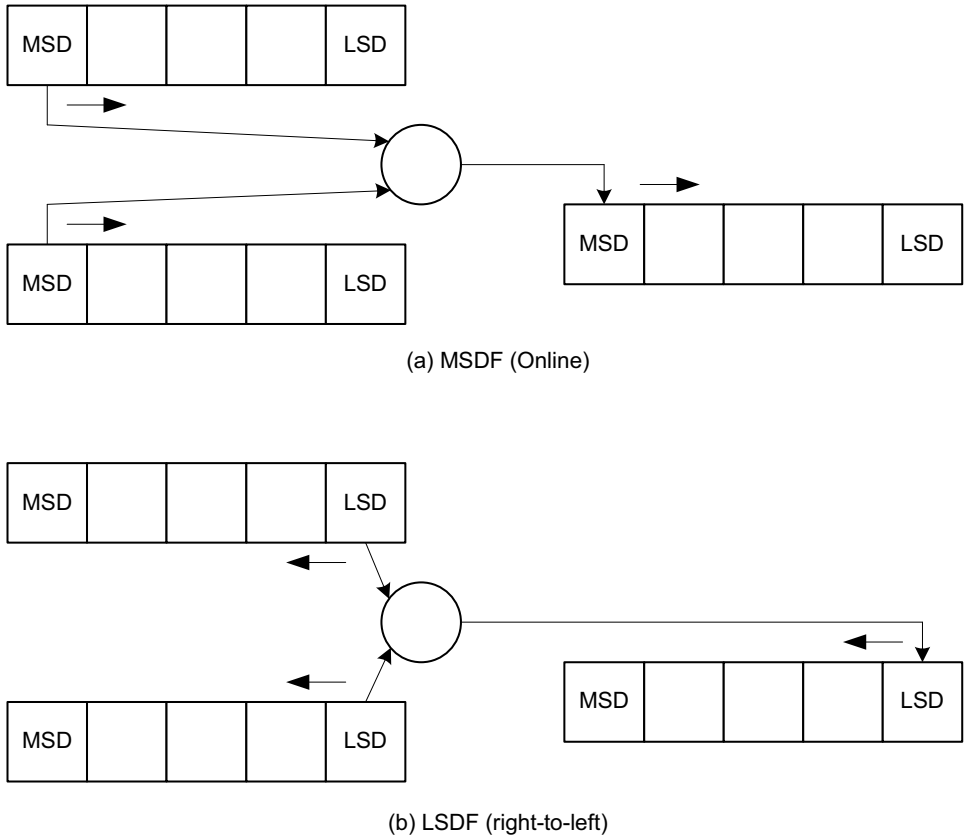


Figure 2.12: Digit flow for different serial arithmetic modes

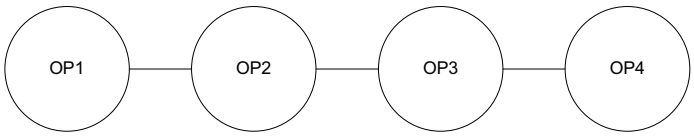
several operations specially those dependent [22] (iii) some operations like division and square rooting calculations cannot be implemented in LSDF mode.

The online delay, or latency δ, is an important characteristic of online arithmetic. It represents the minimum number of input operands digits that are needed to produce the first digit of the result (i.e. the i^{th} digit of the result is produced after $\delta + i$ input digits are received). Figure 2.11 shows the online delay. Even though this additional overhead could be considered as a disadvantage, it can be masked by allowing successive operations to execute in an overlapped manner with a delay of δ cycles [22]. Each operation can start when the first digit of the result of the previous operation appears (i.e. after δ cycles of the beginning of the previous operation). In contrast to conventional parallel approach, online arithmetic does not require completion of an operation before beginning successive operations. Figure 2.13 shows the difference between parallelism in online arithmetic and conventional parallel approaches. According to that, overlapping between arithmetic operations is better and more effective in online arithmetic than in parallel approaches with strong data dependencies. Hence, online arithmetic is more advantageous when more operations are to be performed in sequence [22]. The online delay δ is fixed and depends on the type of the operation itself regardless of the size of inputs (independent of the precision), whereas, LSDF mode has a small fixed initial delay for addition and multiplication but the initial delay for division, square rooting, and max/min operations depends on the size of the input which makes this mode not suitable to these

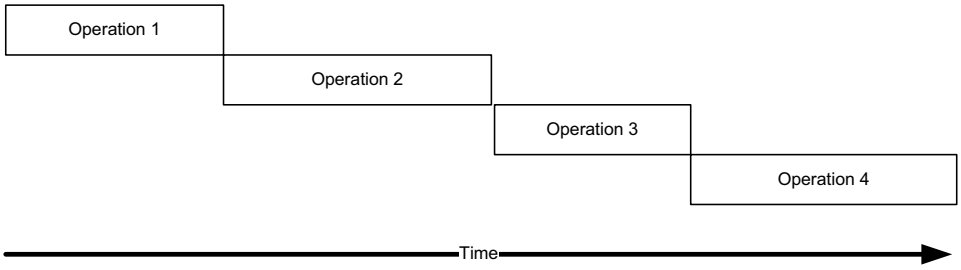
operations.

Since the result is calculated in digit-by-digit manner, online approach has a low-bandwidth communication requirements which is preferable in high concurrent special purpose VLSI designs where the interconnections have a significant impact on the area and time [22].

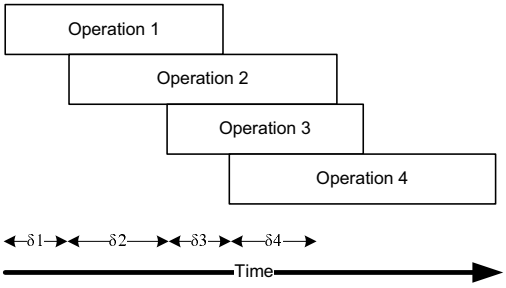
Online (MSDF) mode computation requires flexibility in computing and selecting output digits based on partial information of inputs [22][19]. This flexibility, which could help to introduce compensation in the following digits and limit the carry propagation to only one digit position at most, can be achieved by representing operands digits using redundant number system where multiple representations can represent single value. *Carry-Save* and *signed-digit* are the main redundant number representations in binary systems and can be also used for other radices [19]. Signed-digit redundant number system is the most frequently used in online arithmetic with both symmetric $\{-a, ..., 0, ..., a\}$ and asymmetric $\{-b, ..., 0, ..., c\}$ digit sets. Heterogeneous representations can be used to optimize the implementation of complex computations since different redundant representations are possible. Also, redundant representation of some internal signals may enhance the addition operation [19]. Converting redundant numbers into conventional representation requires carry-propagation addition in parallel arithmetic operations while this conversion can be performed efficiently without carry-propagation addition by using on-the-fly conversion method [19]



(a) series of arithmetic operations



(b) Execution in conventional arithmetic



(c) Execution in online arithmetic

Figure 2.13: Time line for executing sequence of arithmetic operations

2.5.2 Online Multiplication

An online algorithm for multiplication was introduced by Kishor Trivedi and Milos Ercegovac in 1977 [21]. It was derived following the well-known technique of incremental multiplication in addition to the use of redundant number system [21]. The algorithm accepts two serially received n-digit input operands X, Y starting with the most significant digit and produces digits of the output product P in the same manner after the online delay of $\delta = 2$. Both inputs and output are represented as radix-r normalized floating-point fraction in the range $(-1, 1)$ in radix-r redundant number system. Then, X, Y, and P are expressed as [21]:

$$X = \sum_{i=1}^{m} x_i \,.\, r^{-i} \tag{2.20}$$

$$Y = \sum_{i=1}^{m} y_i \,.\, r^{-i} \tag{2.21}$$

$$P = \sum_{i=1}^{m} p_i \,.\, r^{-i} \tag{2.22}$$

Accordingly, we define X_j, Y_j, and P_j as the accumulated digits of X, Y, and P at the j^{th} cycle [22][19]. Thus,

$$X_j = \sum_{i=1}^{j} x_i \,.\, r^{-i} = X_{j-1} + x_j \,.\, r^{-j} \tag{2.23}$$

$$Y_j = \sum_{i=1}^{j} y_i \,.\, r^{-i} = Y_{j-1} + y_j \,.\, r^{-j} \tag{2.24}$$

$$P_j = \sum_{i=1}^{j} p_i \,.\, r^{-i} = P_{j-1} + p_j \,.\, r^{-j} \tag{2.25}$$

The partial product at j^{th} cycle is given by:

$$X_j.Y_j = X_{j-1}.Y_{j-1} + (X_j.y_j + Y_{j-1}.x_j)r^{-j}$$

The error bound at j^{th}cycle is given by [19]

$$|X_j.Y_j - P_j| < r^{-j}$$

Let P_j be a scaled partial product, then:

$$P_j = X_j.Y_j.r^j = rP_{j-1} + (X_j.y_j + Y_{j-1}.x_j) \tag{2.26}$$

Where $P_0 = 0$ [21]. In order to generate the output digits from left-to-right as required, a signed-digit redundant number system must be adopted. Symmetric

redundant digit set need to be used. [21]

$$D_\rho = \{-\rho, -(\rho-1), ..., -1, 0, 1, ..., (\rho-1), \rho\}$$

where

$$\frac{r}{2} \leq \rho \leq r-1$$

The basic recurrence relation (2.26) of online multiplication is given by:

$$w_j = r(w_{j-1} - d_{j-1}) + (X_j.y_j + Y_{j-1}.x_j) \tag{2.27}$$

where the digits $d_j \in D_\rho$ are selected using the following selection function: [21]

$$d_j = SELM(w_j) = sign(w_j).\lfloor |w_j| + 1/2 \rfloor \tag{2.28}$$

From (2.26) and (2.27), the following relation can be derived by induction:

$$w_j = P_j - \sum_{i=1}^{j-1} d_i.r^{(j-i)} \tag{2.29}$$

After rearranging and substituting $n = j$, we have:

$$P_n = X.Y.r^n = r^n \sum_{i=1}^{n-1} d_i.r^{(-i)} + w_n \tag{2.30}$$

or

$$X.Y = \sum_{i=1}^{n} d_i.r^{(-i)} + (w_n - d_n)r^{-n}$$

Since the difference between the w_j and d_j is less than or equal to 1/2, then the quantity $\sum_{i=1}^{n} d_i.r^{(-i)}$ represents the most significant part of the product in redundant form.

Since product digits $d_i \in D_\rho$, the selection function $SELM(w_j)$ generates digits $|d_i| \leq \rho$. To guarantee that, this condition need to be satisfied:

$$|w_j| < \rho + 1/2, \quad \forall j = 1, 2, ..., n \tag{2.31}$$

To ensure that the condition in (2.31) is satisfied, an upper bound M is defined on the values of the inputs X, Y. Let $X, Y \leq M$, $|w_{j-1} - d_{j-1}| \leq 1/2$, and $|x_j|, |y - j| \leq \rho$ then by substituting in (2.27) we get:

$$|w_j| \leq \frac{r}{2} + 2M\rho$$

which, after rearranging and substituting w_j by (2.31), leads to

$$M < \frac{1}{2} - \frac{r-1}{4\rho} \tag{2.32}$$

which means

$$\frac{1}{2r} < M < \frac{1}{4}$$

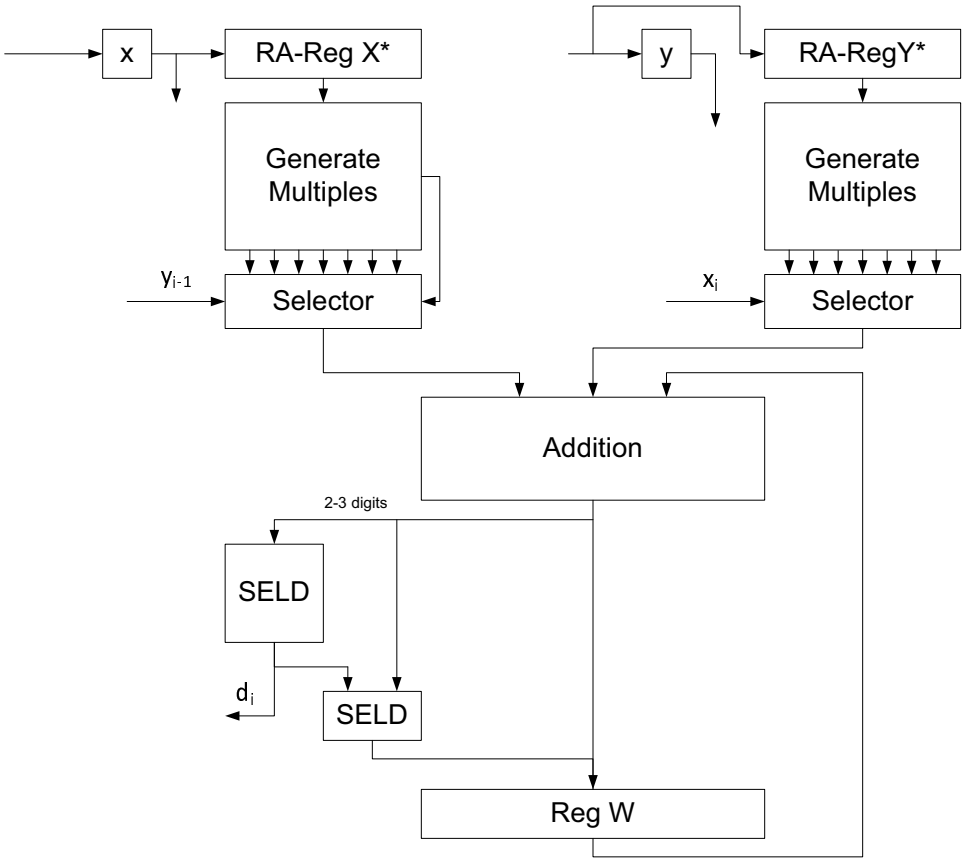
The following example illustrates the online multiplication algorithm for $r = 10$ and $\rho = 5$. Let X and Y be 6-digit decimal numbers, where $X = 0.025472 = 0.03\bar{5}5\bar{3}2$ and $Y = -0.033721 = 0\bar{3}\,\bar{4}3\bar{2}\,\bar{1}$

$P = X.Y = \sum_{j=1}^{6} d_j.10^{-j} + 10^{-6}(w_6 - d_6) = 0.00\bar{1}141 + 10^{-6} \times 0.58688 = -0.000859 + 0.000000058688 = -0.000858941312$

Figure 2.14 shows a possible implementation of the online multiplication algorithm.

Table 2.10: Online Multiplication Example

j	x_j	y_j	$X_j.y_j + Y_{j-1}.x_j$	w_j	d_j	$2(w_j - d_j)$
1	0	0	0	0	0	0
2	3	-3	-0.09	-0.09	0	-0.9
3	-5	-4	0.05	-0.85	-1	1.5
4	5	3	-0.0935	1.4065	1	4.065
5	-3	-2	0.05016	4.11516	4	1.1516
6	2	-1	-0.092912	1.058688	1	0.58688



(*) RA registers appends the new digit to the right to produce the online operands in the conventional represnetation

Figure 2.14: General Implementation of Radix-r Online Multiplier

CHAPTER 3

DECIMAL ONLINE MULTIPLIER DESIGN

The general algorithm of online multiplication is described in Section 2.5.2. To derive the algorithm for radix 10, we replace the variable r which represents the radix by 10. Then, the recurrence relation of the radix 10 algorithm is as follows:

$$w_j = 10 \times (w_{j-1} - d_{j-1}) + (X_j.y_j + Y_{j-1}.x_j) \tag{3.1}$$

where the digits d_j, chosen from a digit set D_ρ, are determined by the following selection function [21]:

$$d_j = SELM(w_j) = sign(w_j).\lfloor |w_j| + 1/2 \rfloor$$

The algorithm followed in the radix-10 online multiplication is given in Figure 3.1.

Online Multiplication Algorithm (radix 10)

Input: $X=\sum_{j=-3}^{n-1} x_j \times 10^{-j-3}$, $Y=\sum_{j=-3}^{n-1} y_j \times 10^{-j-3}$
Output: P_{out} = output digit
1. initialization
    x[-3]= y[-3]= w[-3]= 0
    for j=-3, -2, -1
    x[j+1] ← CA(x[j],x_{j+4})
    y[j+1] ← CA(y[j],y_{j+4})
    v[j+1] = 10×w[j]+(x[j]y_{j+4} + y[j+1]x_{j+4})×10^{-3}
    w[j+1] ← v[j]
    end for
2. recurrence
    For j= 0,…,n-1
    x[j+1] ← CA(x[j],x_{j+4})
    y[j+1] ← CA(y[j],y_{j+4})
    v[j+1] = 10×w[j]+(x[j]y_{j+4} + y[j+1]x_{j+4})×10^{-3}
    p_{j+1} = SELD(v[j])
    w[j+1] ← v[j] - p_{j+1}
    p_{out} ← p_{j+1}
    end For

where:

- n is the operands size in digits
- CA is a conversion and appending function.
- SELD() is the product-digit selection function.
- P_{out} is the product digit output.

Figure 3.1: Radix-10 Online Multiplication Algorithm for Radix 10

3.1 Choice of Digit Set and Encoding

as explained in Section 2.5.2, online algorithms are designed using redundant number systems. Signed-digit and carry-save are the most commonly used redundant number systems. In decimal arithmetic, several SD encodings have been described together with their addition algorithms (Section 2.3). The choice of encoding is an essential step in designing online decimal multiplication hardware since the encoding

affects the digit set used in the design as well as the design complexity.

By referring to the recurrence relation in equation 3.1, two multiplication operations are needed together with three addition and subtraction operations. The required multiplications are digit-by-word decimal multiplications. To perform these multiplications, a set of multiples (primary set or secondary set) need to be generated. The product of each multiplication operation is computed by selecting and adding appropriate multiples. These multiplications are calculated at each iteration since the quantities X_j and Y_{j-1} accept new digit at each cycle. The operation of multiples generation and computation is in the critical path of the online multiplication operation so the choice of digit encoding should both simplify the process of multiples generation and have an efficient addition algorithm in order to enhance the overall performance.

The choice of digit set also affects the complexity of the design since it affects the number of multiples that need to be generated each iteration. As the number of multiples increases, the hardware complexity and components needed to generate these multiples become more complex resulting in larger implementation area and slower speed. Further, with SD set, both positive and negative multiples need to be generated.

To summarize, when a given digit set and encoding, the following criteria have been used:

1. The number system used should be redundant (signed-digit is preferable).

2. A smaller size digit set is preferred in order to reduce the number of multiples to be generated.

3. The encoding should result in an efficient addition algorithm and hardware implementation .

4. The encoding should allow fast and simple generation of multiples (primary set or secondary set of multiples).

5. The encoding should allow fast and simple sign conversion as the digit set is signed.

The decimal SD encodings described together with their addition algorithms in Section 2.3 are evaluated against the above criteria. All unsigned BCD digit sets and encodings can not be chosen as only redundant signed-digit sets are needed. For the [-6,+6] [26], [-7,+7] [28], and [-9,+9] [29] [27] digit sets , the [-6,+6] Svoboda digit set has the smallest size and hence needs the least number of multiples. A smaller digit set, smaller set of multiples calculated, and simpler hardware components. Thus, according to the second criteria, designs with digit set [-9,+9] are ruled out. Also the hardware design of algorithm described in [27] is more complicated with inefficient area utilization.For sign representation, algorithms reported in [28] and [29] uses 2's complement representation requiring additional correction steps to generate transfer quantities during the multiple generation process compared with the Svoboda encoding [26]. Although multiplication by 2 in 2's complement system

can be accomplished by left-shifting, this step is not enough in SD representation. For example, some multiplication operations require both sign conversion and generation of a transfer digit (e.g. in digit set [-7,+7] $4 \times 2 = 8 = 1\overline{2}$, the input (4) is positive (0100) while the output (-2) is negative (1110) with a transfer digit of (+1)). Thus, it is important that the selected digit set and encoding have simple sign conversion mechanism.

Based on these criteria and the above discussion, the Svoboda encoding (described in [26]) has been chosen for our design from amongst other encodings discussed in Section 2.3. The digit set used in the Svoboda design has been modified to be from (-5) to (+5) in order to simplify multiples generation and sign detection. With the restricted [-5,+5] modified digit set, the most significant bit determines the sign of the digit rather than through the use of a parity circuit. Table 3.1 shows the adopted modified Svoboda codes. The modified [-5,+5] digit set is used for the inputs and for multiples generation while the original [-6,+6] digit set is used in the addition and reduction operations.

3.2 Internal Multiplication Operations

The recurrence relation of online multiplication includes two digit-by-word multiplications ($y_j X_j$ and $x_j Y_{j-1}$). In each iteration, two new digits (x_j and y_j) are received, and the words (X_j and Y_j) are updated. Thus, the two multiplication operations

Table 3.1: Modified Svoboda Signed-digit Codes

x_i	Binary Code					X_i
	e	d	c	b	a	
+5	0	1	1	1	1	+15
+4	0	1	1	0	0	+12
+3	0	1	0	0	1	+9
+2	0	0	1	1	0	+6
+1	0	0	0	1	1	+3
+0	0	0	0	0	0	0
-0	1	1	1	1	1	31
-1	1	1	1	0	0	28
-2	1	1	0	0	1	25
-3	1	0	1	1	0	22
-4	1	0	0	1	1	19
-5	1	0	0	0	0	16

need to be calculated every cycle. Generating all multiples (*primary set*) is not an appropriate choice since it increases the clock period and the implementation area. Generating a subset of the multiples, *a secondary set*, such that all other multiples can be calculated as the sum of two elements of the secondary set. Two different secondary sets have been considered for our design: the first one consists of the multiples $\{-A, A, 2A, 5A\}$ and the second one consists of the multiples $\{A, 2A, 4A\}$. Generating other multiples from these sets is described in Table 3.2.

Table 3.2: Use secondary set to generate all multiples

Digit	Secondary Set $(A, , -A, 2A, 5A)$	Secondary Set $(A, 2A, 4A)$
5	$5A$	$A, 4A$
4	$5A, -A$	$4A$
3	$A, 2A$	$A, 2A$
2	$2A$	$2A$
1	A	A
0	0	0

The choice of multiples is based on the absolute value of the multiplier digit and then the sign of the result is changed if the multiplier digit is negative. Changing the sign in Svoboda encoding is simply done by bit complementation.

Multiplying a Sovoboda encoded digit ($X = x_4x_3x_2x_1x_0$) by 2 is simply done by shifting one bit to the left with feedback coming from the second most significant bit ($2X = x_3x_2x_1x_0x_3$, $t = (-1)^{x_4}(x_4 \oplus x_3)$). A transfer bit having a weight of 10 is sent to the higher order digit as (+1 or -1) if the most significant two bits are different and the sign of the transfer is determined by the sign of the current digit (the most significant bit). The transfer bits coming from lower significant digit are then added to the shifted number using a 4-bit ripple carry adder. The digit after shifting is even (i.e. 0,2,4,-2,-4) which means the output digit will be in the range [-5,+5] after adding the transfer bits. Figure 3.2 shows the operation of multiplying by 2.

The input carry to the first full adder is estimated by combinational logic circuit based on the input digit. The addition circuit is similar to the part of the Svoboda adder circuit that adds the interim sum and the transfer digit. The logic function of this carry is ($c_{in} = t^+_{in}t^-_{in} + x_1\overline{x_0}t^-_{in} + x_3t^-_{in} + x_4t^-_{in} + x_4x_3x_1t^+_{in}$). The transfer bits are given by ($t^-_{out} = x_4\overline{x_3}$) and ($t^+_{out} = \overline{x_4}x_3$) where input ($X = x_4x_3x_2x_1x_0$) and t^+_{in} and t^+_{in} are the input transfer bits.

Multiplying a Svoboda encoded digit by 4 is achieved by multiplying the digit by 2 twice. The same procedure used to multiply by 2 is applied twice to get the

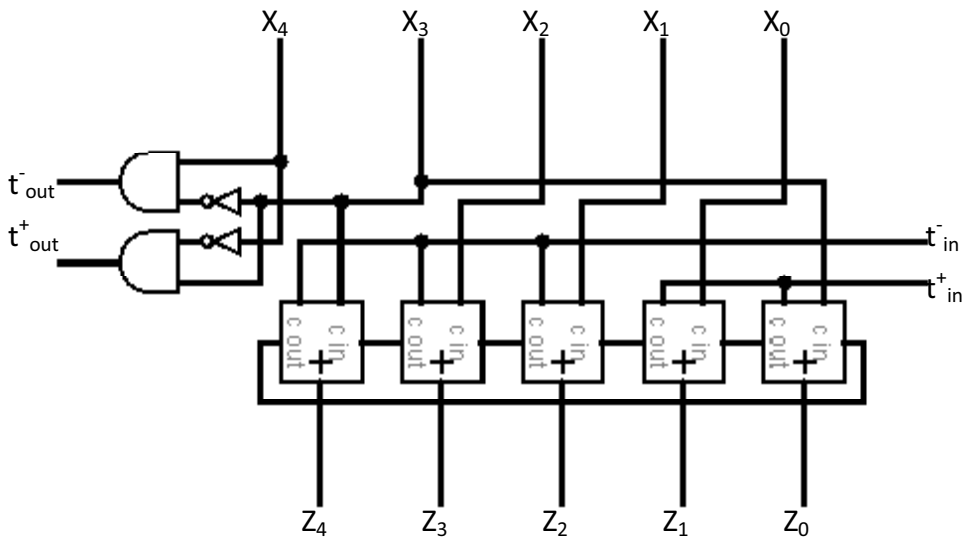


Figure 3.2: Multiplying a Svoboda encoded digit by 2

result of multiplication by 4.

Multiplying a Svoboda encoded digit by 5 is achieved by shifting the whole number by one digit to the left (multiplying by 10) then dividing the result by 2. As an example consider the number ($-25 = \overline{3}5$); after multiplication by (10) it is equal to ($\overline{3}50$). The operation of division by 2 is achieved by the two level combinational logic shown in Figure 3.3. After the division operation, a transfer digit of value (+5 or -5) is sent to the lower order digit if the current digit is odd (e.g. dividing $\overline{3}$ results in $\overline{1}$ in the currect position and (-5) is sent to lower order digit. Also the result of division could be ($\bar{2}$) in the current position and (+5) is transfered to lower order digit). The least significant bit of the digit is used to distinguish between even numbers and odd numbers. The sign of the transfer digit is determined by the sign of lower order digit to avoid carry propagation. When the signs of two consecutive

digits are the same, the sign of the transfer digit is converted and (+1) or (-1) is added to the current digit based on the sign. On the other hand, a transfer back digit with the same sign of current digit (+5 or -5) is sent without any addition if the signs are different (i.e. the sign of ($\bar{3}$) and (5) are different then the result of dividing ($\bar{3}$) by 2 is ($\bar{1}$) and (-5) is transfered back to lower order digit whereas when (5) is divided by 2, the result is (3) and (-5) is transfered back to lower order digit since the signs of (5) and (0) are both positive). Multiplying the number ($\bar{3}5$) by (5) gives ($\bar{3}50$) after shifting the whole number by one digit to the left and then ($\bar{1}\bar{2}\bar{5} = -125$) after dividing by 2. The resultant digit after dividing the digit by 2 is in the range [-3,3]. Making the sign of the transfered-back digit different from the sign of the current digit avoids further transfer bits (i.e. the transfer back digit equals to (+5) when the current digit is negative and vice versa). Figure 3.4 shows an example of multiplying a Svoboda encoded number by 5. The flowchart of multiplication-by-5 operation is shown in Figure 3.5. The operation of dividing a Svoboda encoded digit by two can be performed using a two level of combinational logic according to the following Boolean equations:

$$Z_i = \begin{cases} z_4 = \overline{x_0 x_3} x_4 + \overline{x_0} x_3 \overline{x_4} + x_0 \overline{x_1 x_2} + x_0 x_1 x_2 \\ z_3 = \overline{x_0 x_3} x_4 + \overline{x_0} x_2 + x_1 x_2 \\ z_2 = x_1 x_2 + x_0 (x_3 \oplus x_4) \\ z_1 = x_0 \\ z_0 = x_0 \end{cases}$$

Table 3.3 shows all digits in the Svoboda digit set with the result of multiplying by 2 and dividing by 2.

Table 3.3: Svoboda encoded digits and their results of multiplication by 2 and division by 2

Digit	Multiplied by 2		Divided by 2	
	T_{out}	D	D	Transfer back
5	T_{out}^{+}	0	2 3	+5 -5
4	T_{out}^{+}	-2	2	0
3	T_{out}^{+}	-4	1 2	+5 -5
2	0	4	1	0
1	0	2	0 1	+5 -5
0	0	0	0	0
-1	0	-2	0 -1	-5 +5
-2	0	-4	-1	0
-3	T_{out}^{-}	4	-1 -2	-5 +5
-4	T_{out}^{-}	2	-2	0
-5	T_{out}^{-}	0	-2 -3	-5 +5

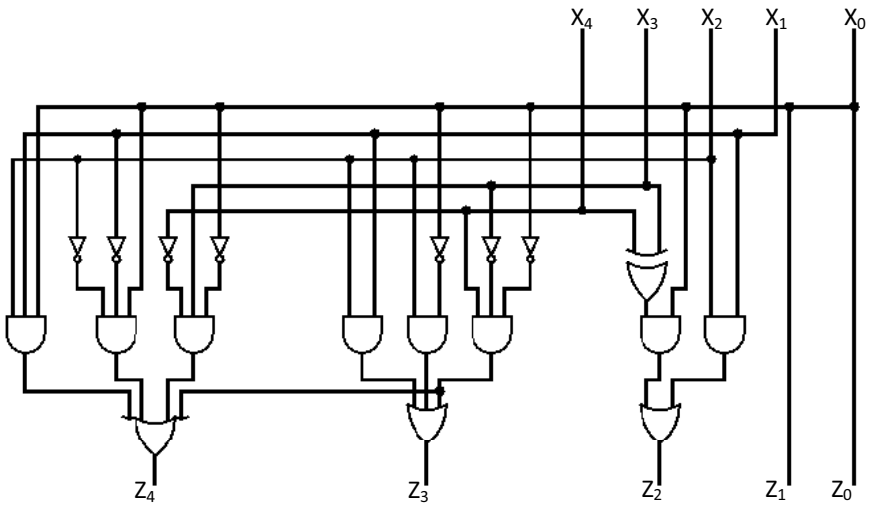


Figure 3.3: Dividing a Svoboda encoded digit by 2

3.3 Hardware Architecture

The recurrence relation of decimal online multiplication algorithm is

$$w_j = 10 \times (w_{j-1} - d_{j-1}) + (X_j.y_j + Y_{j-1}.x_j)$$

Then, the data path of decimal online multiplication algorithm should perform the following steps each iteration:

- Generate secondary set of multiples $((-A, A, 2A, 5A)$ or $(A, 2A, 4A))$.
- From the secondary set, select the combination of multiples to add so as to calculate $X_j y_j$ and $Y_{j-1} x_j$.
- Calculate the sum $y_j X_j + x_j Y_{j-1}$.

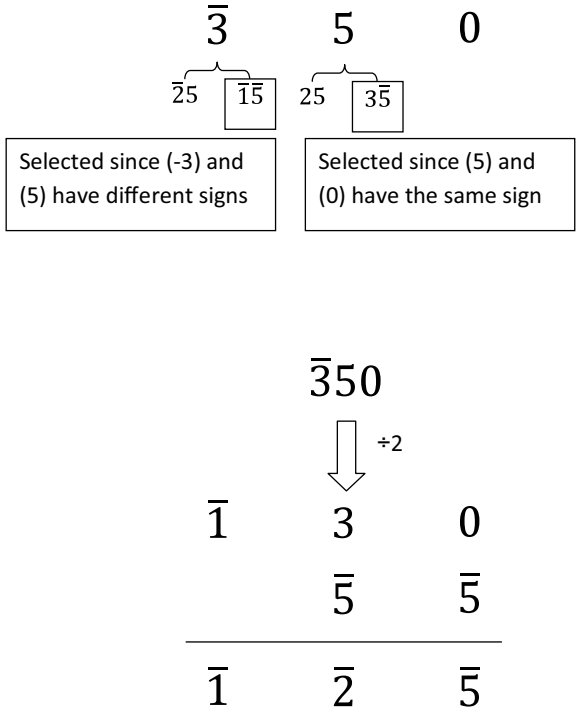


Figure 3.4: Example of multiplying a Svoboda encoded number by 5

- Compute the value of w_j by adding $(w_{j-1} - d_{j-1} \times 10)$ calculated in the previous iteration.

- Derive the output digit d_j according to the selection function $d_j = SELM(w_j) = sign(w_j).\lfloor |w_j| + 1/2 \rfloor$.

- Compute the quantity $10 \times (w_{j-1} - d_{j-1})$ and store it for use in the next iteration.

The first two steps depend on the adopted secondary set of multiples used. In this design, two sets of multiples, $(-A, A, 2A, 5A)$ and $(A, 2A, 4A)$, have been implemented. The data path of each one is described below.

In the first implementation, the set of chosen multiples are $(-A, A, 2A, 5A)$. The complete data path is shown in Figure 3.7. Following is a descriptive of the data path modules used in the design:

Right append register: This register accepts one digit at each clock cycle and appends it to the right of the already received digits. Two n-digit registers of this type are needed in decimal online multiplication.

2-by-1 Multiplexers: Used to choose between multiples. The output of the multiplexer is zero if the enable signal is not active.

Decimal adder: This adder adds two n-digits Svoboda-encoded decimal numbers. Each adder consists of k digit-by-digit adders that are connected together as shown in Figure 3.6. The internal logic of a digit-by-digit adder is shown in Figure 2.8. The input carries for both stages are generated separately by lookahead logic according to the following Boolean equations:

$$c1_{in} = x_1y_2y_1y_0 + x_2y_3y_2 + x_3y_3 + x_3x_1y_1 + x_3x_2y_2$$

$$c2_{in} = t^+_{in}t^-_{in} + x_1\overline{x_0}t^-_{in} + x_3t^-_{in} + x_4t^-_{in} + x_4x_3x_1t^+_{in}$$

Sign converter: Can change the sign of an input number by complementing all bits. Its functionality is like multiplying the input by the sign.

Multiply-by-2 component: It accepts a n-digit Svoboda-encoded number X and generates $2X$ in (k+1) digits. The process of multiplication is described in the previous section.

Multiply-by-5 component: It accepts a n-digits Svoboda-encoded number X and generates $5X$ in (k+1) digits. The process of multiplication is described in the previous section.

Output digit selection (SELD): This component accepts 2 most significant digits of (w_j) and calculates the output digit d_j. It computes the formula $d_j = sign(w_j).\lfloor |w_j| + 1/2 \rfloor$. It includes circuitry to detect the sign, circuitry to apply the floor function, and an adder (not a complete adder) to increment the output digit by one when needed.

Register (W): Used to hold the value of W from one cycle to another.

Zeroing Logic: This component is used to either pass the input value or force all output bits to zeros. This component is used in the architecture with secondary set includes (A, $2A$, $4A$) to pass the value of (A) or zeros.

The difference between the first and second implementations is in the set of used multiples. Using different multiple set (secondary set) affects the count and the type of components used. The second implementation uses a secondary set that includes $(A, 2A, 4A)$ which means that there is no need to use Multiply-by-5 component. The

generation of $4A$ is achieved by using two Multiply-by-2 components consecutively. The data path of this implementation is shown in Figure 3.8.

As mentioned before, the digit set of the output of the components that generate multiples is [-5,+5]. After signed-digit addition, the resulting digit set is the regular [-6,+6] Svoboda digit set. The online algorithm guarantees that the output digit is in the Modified digit set [-5,+5].

Table 3.4 shows a snapshot of the online multiplication operation. Inputs are ($X = 0.0422996 = 0.042300\bar{4}$) and ($Y = 0.2617661 = 0.3\bar{4}2\bar{2}\bar{3}\bar{4}1$).The output digits so far are (000) The snapshot is taken at 4^{th} cycle.

Table 3.4: Snapshot of the designs during the

	Architecture 1 $(A, -A, 2A, 5A)$	Architecture 2 $(A, 2A, 4A)$
Input digits	$x_4 = 3$, $y_4 = \bar{2}$	
Accumulated inputs	$X_4 = 0423$ $Y_3 = 3\bar{4}2$	
Multiples generated (X)	$X = 0423$ $-X = \bar{0}\bar{4}\bar{2}\bar{3}$ $2X = 01\bar{2}5\bar{4}$ $5X = 0212\bar{5}$	$X = 0423$ $2X = 01\bar{2}5\bar{4}$ $4X = 02\bar{3}\bar{1}2$
multiples selected	$2X$	$2X$
$y_4 \times X_4 = (-2) \times X_4$	$0\bar{1}2\bar{5}4$	$0\bar{1}2\bar{5}4$
Multiples generated (Y)	$Y = 3\bar{4}2$ $-Y = \bar{3}4\bar{2}$ $2Y = 1\bar{5}24$ $5Y = 1310$	$Y = 3\bar{4}2$ $2Y = 1\bar{5}24$ $4Y = 105\bar{2}$
multiples selected	$Y + 2Y$	$Y + 2Y$
$x_4 \times Y_3 = (3) \times Y_3$	$001\bar{2}\bar{1}\bar{4}$	$001\bar{2}\bar{1}\bar{4}$
$y_4X_4 + x_4Y_3$	$001\bar{3}014$	$001\bar{3}014$
$10 \times (w_3 - d_3)$	110040	110040
$W = y_4X_4 + x_4Y_3 + 10 \times (w_3 - d_3)$	$111\bar{3}414$	$111\bar{3}414$
SELD(W)	1	1
$10 \times (w_4 - d_4)$	$11\bar{3}414$	$11\bar{3}414$

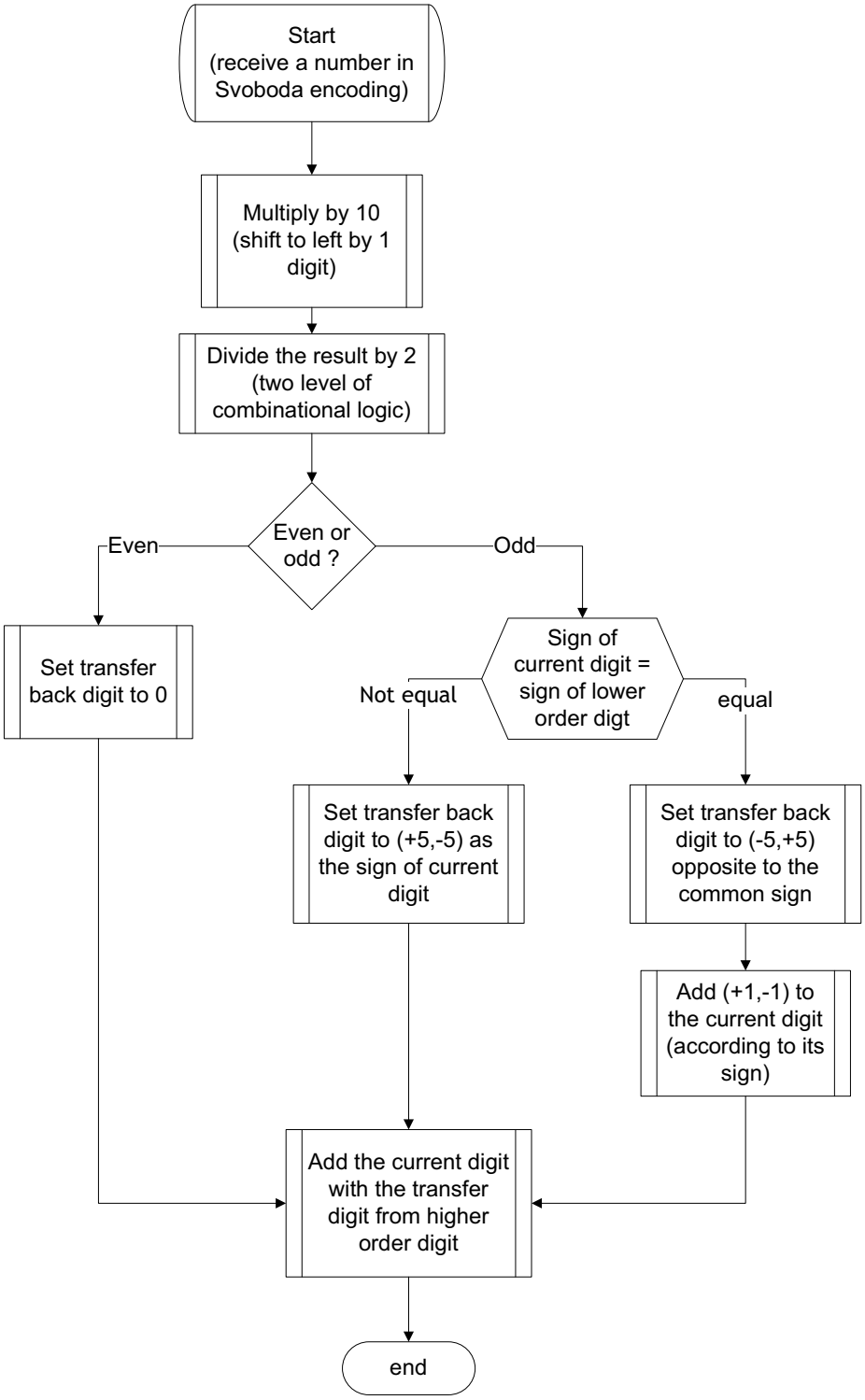
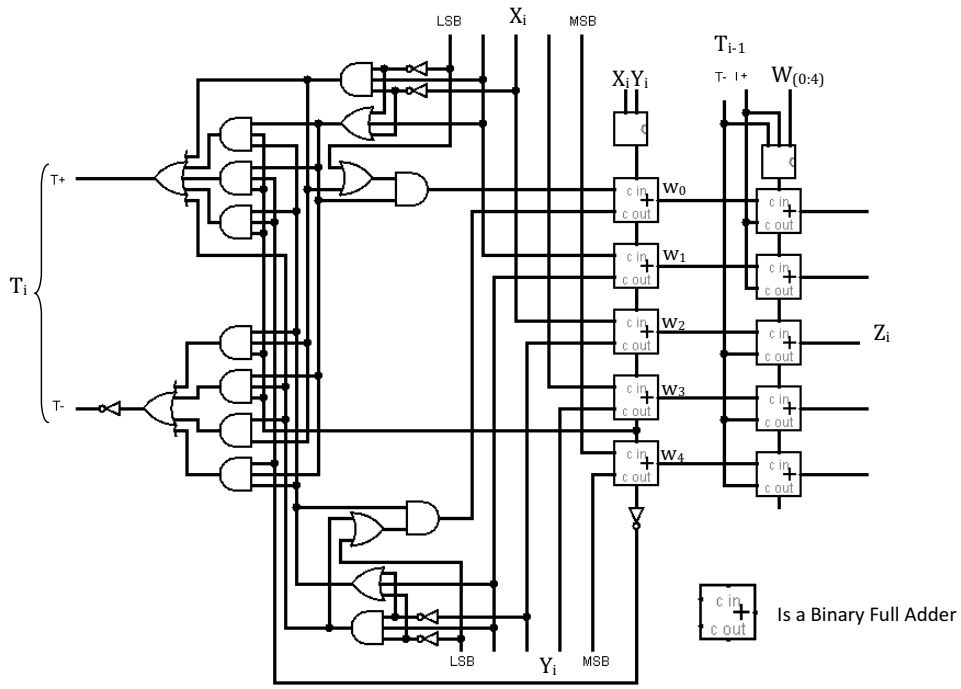
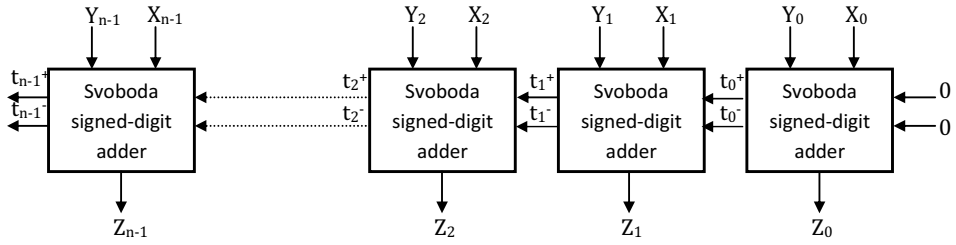


Figure 3.5: Flowchart for Multiplying-by-5 Operation

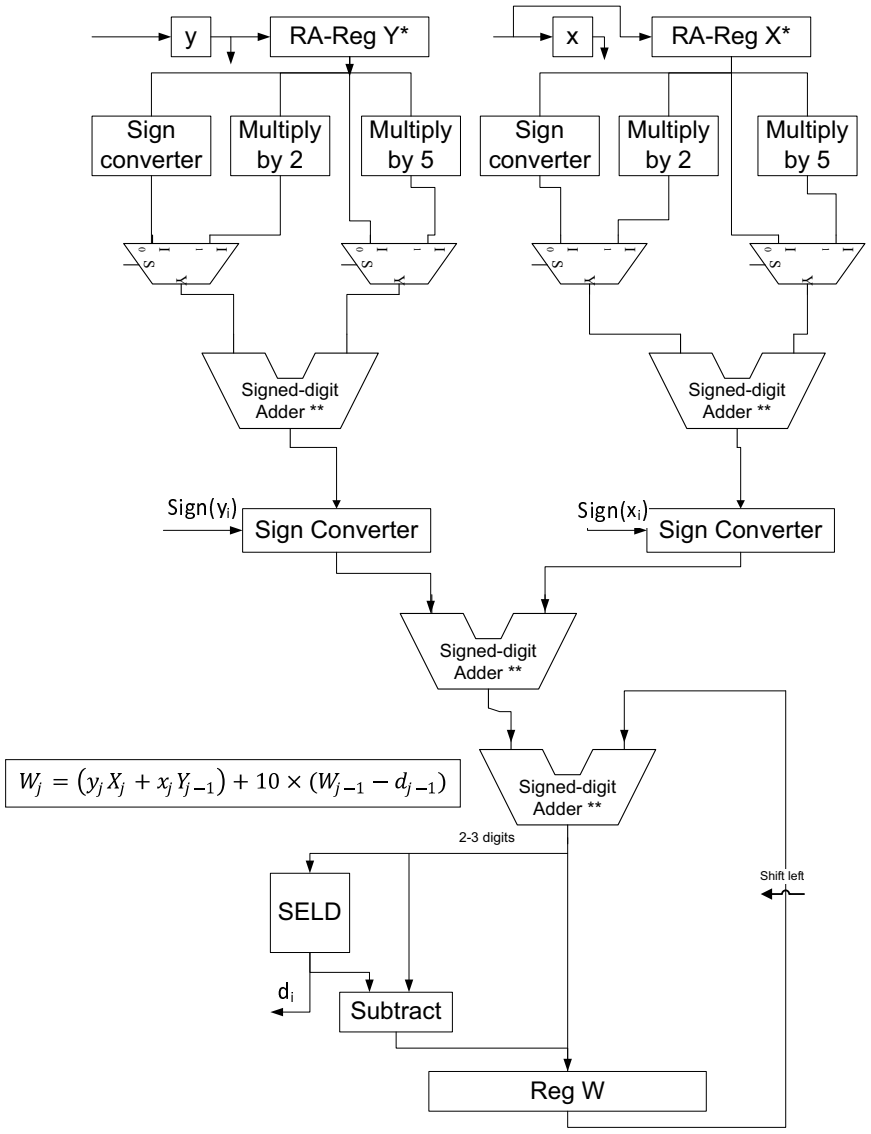


(A)



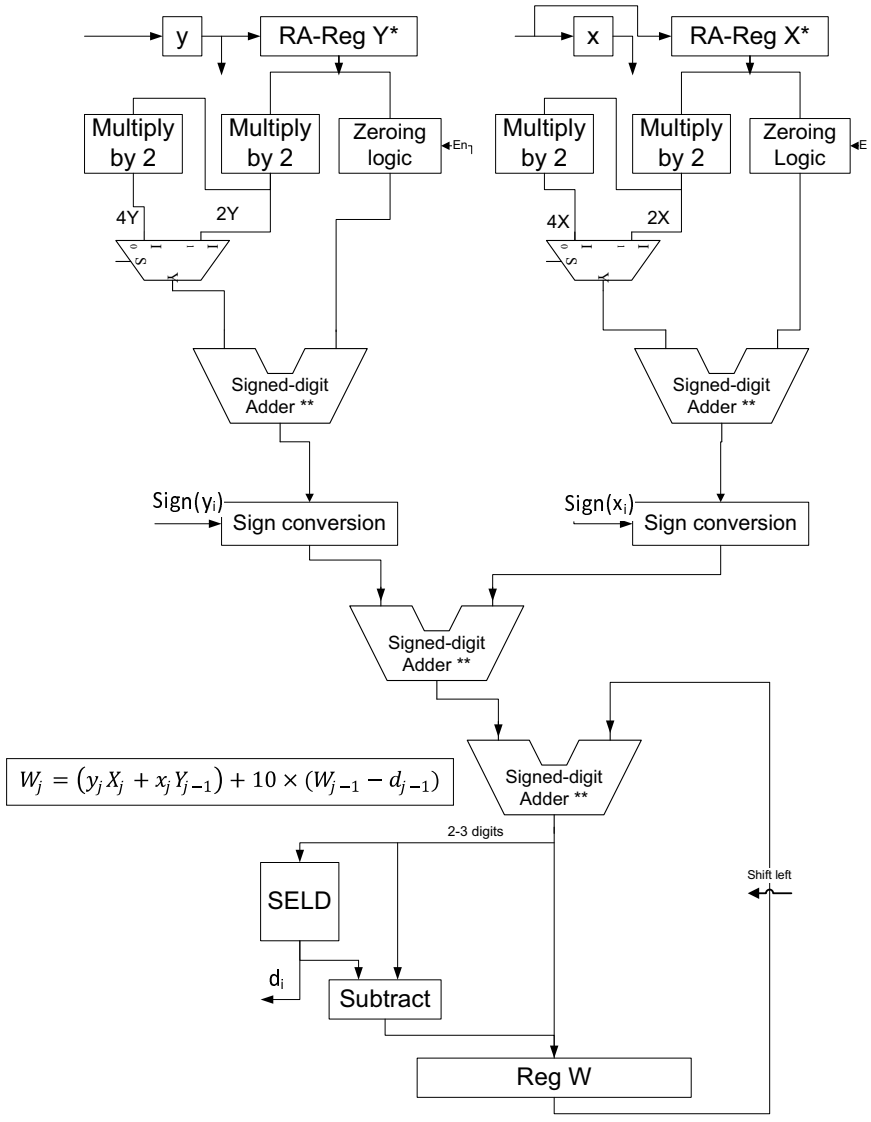
(B)

Figure 3.6: Svoboda Signed-digit Adder: (A) Single digit adder, (B) n-digit adder



(*) RA registers appends the new digit to the right to produce the online operands in the conventional representation
(**) n-digit signed-digit adder which adds two n-digit Svoboda encoded numbers

Figure 3.7: Data Path of Decimal Online Multiplier (secondary set of multiples (-A,A,2A,5A))



(*) RA registers appends the new digit to the right to produce the online operands in the conventional representation
(**) n-digit signed-digit adder which adds two n-digit Svoboda encoded numbers

Figure 3.8: Data Path of Decimal Online Multiplier (secondary set of multiplies (A,2A,4A))

CHAPTER 4

SYNTHESIS RESULTS AND DISCUSSION

4.1 Synthesis Results and Optimization

The design of online decimal multiplier has been modeled using VHDL hardware description language and synthesized on XILINX FPGA. Two different datapath architectures have been modeled structurally; one using a secondary set of $(-A, A, 2A, 5A)$ and the other using secondary set of $(A, 2A, 4A)$. The developed models have been synthesized once for delay optimization and another for area optimization. Xilinx ISE 6.2i was used to synthesize designs on VertexE xcv3200e FPGA. The whole design is bounded between two registers with no combinational logic before or after these registers. Thus, the synthesized longest propagation delay only affects

the clock period (the longest path is bounded between two registers). The synthesis results for the first datapath that uses a secondary set of $(-A, A, 2A, 5A)$ is shown in tables 4.1 and 4.2. Table 4.1 shows the area and minimum clock period for the design when synthesized for area optimization while table 4.2 shows the area and minimum clock period for the design when synthesized for speed optimization. Likewise, synthesis results for the second datapath that uses the secondary set $(A, 2A, 4A)$ is shown in tables 4.3 and 4.4. Table 4.3 shows the area and minimum clock period for the design when synthesized under area optimization while table 4.4 shows the area and minimum clock period for the design when synthesized under speed optimization.

Figure 4.1 and 4.2 show plots of these results where the architecture using the secondary set $(-A, A, 2A, 5A)$ is designated by (Mul125) while the architecture using the secondary set $(A, 2A, 4A)$ is designated by (Mul124). Figure 4.1 shows that the architecture (Mul124) gives better area compared to (Mul125) architecture when synthesized for area optimization since the combinational logic used in multiple generation is simpler. As described in section 3.2, the operation of multiplying a Svoboda-encoded number by 4 involves two stages of shifting and addition operation using 4-bit ripple carry adder while the operation of multiplying by 5 involves some combinational logic and two 4-bit ripple carry additions. This also explains the fast rate of area increase in (Mul125) architecture compared to that of the (Mul124) architecture. When synthesized for optimum delay, the design area for both ar-

Table 4.1: Delay and Area Synthesis Results for the First Implementation of Online Decimal Multiplier $(-A, A, 2A, 5A)$ (Area optimization)

Number of Digits	Area (in slices)	Period (in ns)	Logic delay (in ns)	Routing Delay (in ns)
7	533	67.768	20.192	47.576
8	569	66.910	19.794	47.116
16	892	66.890	19.794	47.096
17	934	66.890	19.794	47.096
25	1247	66.890	19.794	47.096
26	1292	66.890	19.794	47.096
34	1587	66.890	19.794	47.096
35	1640	66.910	19.794	47.116
43	1937	66.910	19.794	47.116
44	1980	66.910	19.794	47.116

Table 4.2: Delay and Area Synthesis Results for the First Implementation of Online Decimal Multiplier $(-A, A, 2A, 5A)$ (Speed optimization)

Number of Digits	Area (in slices)	Period (in ns)	Logic delay (in ns)	Routing Delay (in ns)
7	1011	41.754	11.038	30.716
8	1025	42.772	11.716	31.056
16	1450	43.236	10.920	32.316
17	1475	43.440	10.920	32.520
25	1799	43.506	11.318	32.188
26	1899	41.622	11.318	30.304
34	2343	42.498	10.522	31.976
35	2252	43.618	11.318	32.300
43	2400	42.659	11.255	31.404
44	2438	42.958	11.318	31.640

Table 4.3: Delay and Area Synthesis Results for the Second Implementation of Online Decimal Multiplier $(A, 2A, 4A)$ (Area optimization)

Number of Digits	Area (in slices)	Period (in ns)	Logic delay (in ns)	Routing Delay (in ns)
7	431	62.386	18.202	44.184
8	441	62.186	18.202	43.984
16	598	63.682	18.718	44.964
17	614	62.548	18.320	44.228
25	752	63.118	18.718	44.400
26	782	62.838	18.718	44.400
34	918	62.838	18.718	44.120
35	928	63.118	18.718	44.400
43	1072	63.118	18.718	44.400
44	1119	63.118	18.718	44.400

Table 4.4: Delay and Area Synthesis Results for the Second Implementation of Online Decimal Multiplier $(A, 2A, 4A)$ (Speed optimization)

Number of Digits	Area (in slices)	Period (in ns)	Logic delay (in ns)	Routing Delay (in ns)
7	857	43.014	11.318	31.696
8	856	43.068	11.716	31.352
16	998	41.936	10.920	31.016
17	1019	41.960	10.920	31.040
25	1174	41.080	10.920	30.160
26	1176	42.016	10.920	31.096
34	1326	41.800	10.920	30.880
35	1343	41.880	10.920	30.960
43	1489	41.936	10.920	31.016
44	1543	42.016	10.920	31.096

chitectures almost doubled since the synthesis tool duplicates some components to reduce the routing delay. Synthesis results show that around more than 70% of the delay is due to routing. Further, results show that the delay due to logic is almost constant (difference between maximum and minimum logic delay is less than 1 ns). This expected since there is no carry propagation through the whole number (signals only propagate to the next digit only). Figure 4.2 plots the clock period for all synthesis results.

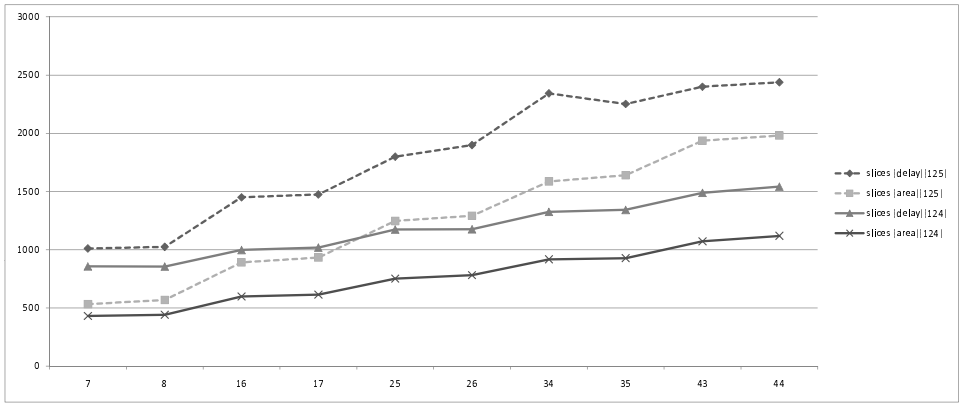


Figure 4.1: Design Area (Number of slices) Vs. Number of digits

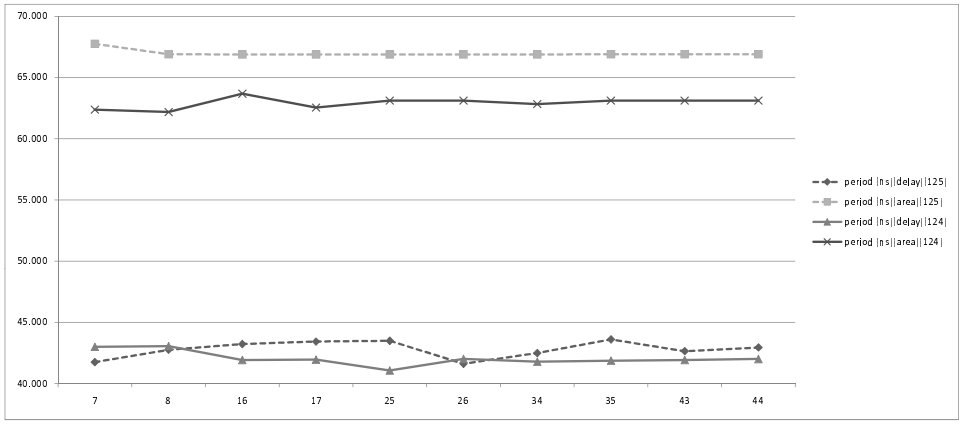


Figure 4.2: Clock period (in nanoseconds) Vs. Number of digits

4.2 Performance Evaluation

Since no other decimal online multiplier has been reported in the literature, thus for, we will compare our results to those of the decimal multiplier reported by Mark A. Erle, Eric M. Schwarz, and Michael J. Schulte in [14]. A non-pipelined version of this sequential multiplier was VHDL modeled and synthesized for this purpose. This multiplier accepts inputs in regular BCD encoding, converts them into [-5,+5] signed-magnitude digits, and generates the partial products. The multiplier uses signed-digit number system with digit set [-5,+5] in order to reduce the cost of combinational logic used for partial products generation. In the reduction stage, the design utilizes the Svoboda adder described in [26]. The multiplication operation begins from the least significant digit of the multiplier. Synthesis results are shown in table 4.5 for area optimization and Table 4.6 for speed optimization.

Figure 4.3 shows a area plot for synthesis results of all designs. The graph includes the number of slices of both architectures (Mul125 and Mul124) together with the sequential multiplier of Erle when synthesized for area optimization. Also Figure 4.4 shows maximum clock period for these designs.

From the figures, we can notice that the clock period of the sequential multiplier is less than that of the online multiplier. This is because the online multiplier uses more components in order to generate the output digit while the sequential multiplier does not need any additional logic to produce output since full inputs operands are

Table 4.5: Delay and Area Synthesis Results for the Sequential Decimal Multiplier (Erle Design) (Area Optimization)

Number of Digits	Area (in slices)	Period (in ns)	Logic delay (in ns)	Routing Delay (in ns)
7	857	43.014	11.318	31.696
8	856	43.068	11.716	31.352
16	998	41.936	10.920	31.016
17	1019	41.960	10.920	31.040
25	1174	41.080	10.920	30.160
26	1176	42.016	10.920	31.096
34	1326	41.800	10.920	30.880
35	1999	45.864	12.512	33.352
43	2315	46.208	12.512	33.696
44	2365	46.288	12.512	33.776

Table 4.6: Delay and Area Synthesis Results for the Sequential Decimal Multiplier (Erle Design) (Speed optimization)

Number of Digits	Area (in slices)	Period (in ns)	Logic delay (in ns)	Routing Delay (in ns)
7	607	32.688	8.532	24.156
8	750	33.298	8.134	25.164
16	1547	31.252	8.532	22.720
17	1652	30.936	8.532	22.404
25	2786	32.172	8.532	23.640
26	2931	32.252	8.532	23.720
34	4599	33.190	8.350	24.840
35	4716	32.767	8.071	24.696
43	4315	31.634	8.134	23.500
44	4404	31.634	8.134	23.500

available. Since both multipliers (sequential and online) use signed-digit number system, the logic delay is almost constant since there is no carry propagation.

The sequential multiplier differs from online multipliers in number of cycles required to generate the complete result. Sequential multiplier requires n cycles to produce the complete result whereas online multipliers require $2n + \delta$ to produce the complete result where δ is the online delay. Although the online multiplier can calculate the complete product after $n + \delta$ cycles, the property of online arithmetic that forces online multipliers to produce one digit every cycle increases the delay.

Figure 4.3 shows that the sequential multiplier has a much higher rate of area increase compared with the online ones. This is because the sequential multiplier uses digit-by-digit multiplication approach using combinational logic rather than the digit-by-word approach used in the online multiplier to generate the partial products. This result shows that digit-by-word multiplication approach may help to reduce area.

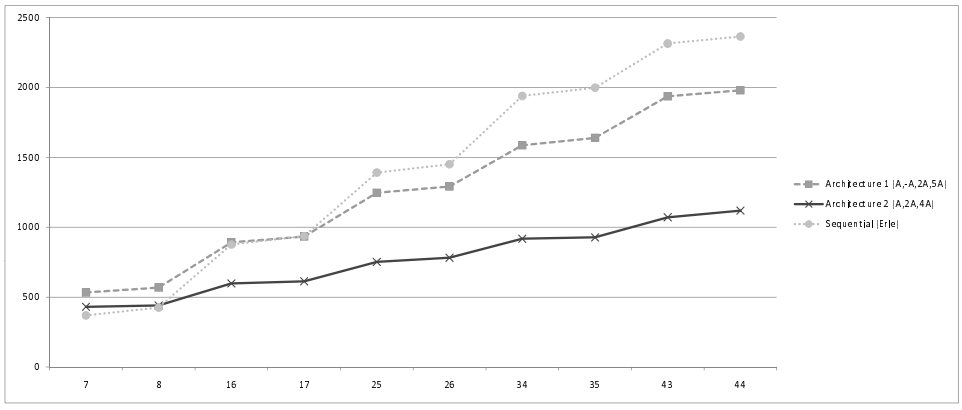


Figure 4.3: Design Area (Number of slices) Vs. Number of digits (Sequential Vs. Online)

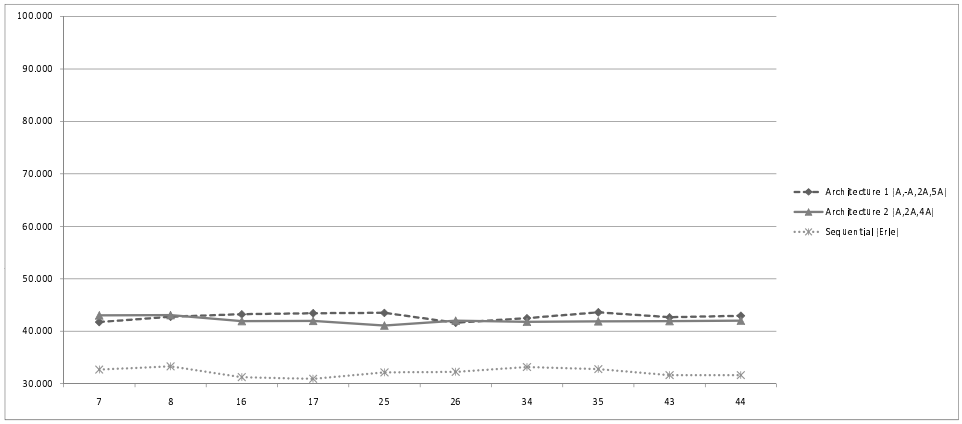


Figure 4.4: Clock period (in nanoseconds) Vs. Number of digits (Sequential Vs. Online)

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

One of the major contributions of the thesis is the design of the first decimal online multiplier. This multiplier computes the product of two decimal number $P = A \times B$ given that A, B, and P are all online inputs and output.

WE have implemented our proposed decimal multiplier architecture for two secondary sets with the corresponding variations in data paths. The proposed implementation of the online decimal multiplier has two datapath designs with two different secondary set of multiples $(A, -A, 2A, 5A)$ and $(A, 2A, 4A)$. The digits are represented in signed-digit redundant representation and encoded with a special encoding proposed by Antonin Svoboda in 1969.

To verify the correctness of the proposed design and estimate the area and speed

of the proposed design, we have developed a VHDL model for our design. The model is parameterizable to allow synthesis with different number of digits. Both data path designs were modeled and synthesized for area and speed optimization. Xilinx ISE 6.2i was used to synthesize the designs on VertexE xcv3200e FPGA.

Synthesis results show that the architecture using the secondary set $(A, 2A, 4A)$(Mul124) is more area efficient and has better delay over the architecture using the secondary set $(-A, A, 2A, 5A)$(Mul125) in general.

The following are some possible future directions for further research related to this work:

- Optimize the encoding of the decimal digits. The encoding should speed-up the addition operation and simplify the generation of multiples operation.

- Develop a novel design for generating multiples that can reduce the area required for this step. The new design does not need to recompute the multiple of the whole received word. Only the multiple of the new incoming digits is computed and used to update the previously computed multiple.

Bibliography

[1] Alvaro Vazquez Alvarez, High-Performance Decimal Floating-Point Units, Phd. Dissertation Jan. 2009.

[2] A. Vazquez, E. Antelo, and P. Montuschi, A New Family of High Performance Parallel Decimal Multipliers, in 18th IEEE Symposium on Computer Arithmetic, June 2007, pp 195204.

[3] M. Cowlishaw, IBM General Decimal Arithmetic Website. http://www.speleotrove.com/decimal/

[4] Mark A. Erle, Algorithms and Hardware Designs for Decimal Multiplication, pp. 217, Lehigh University, November 2008.

[5] W. S. Brown and P. L. Richman, The Choice of Base, Communications of the ACM, Vol. 12 10, pp 560 561, ACM Press, October 1969.

[6] Werner Buchholz, Fingers or Fists? (The Choice of Decimal or Binary representation), Communications of the ACM, Vol. 2 12, pp311, ACM Press, December

1959.

[7] Michael F. Cowlishaw, Decimal Floating-Point: Algorism for Computers, Proceedings of the 16th IEEE Symposium on Computer Arithmetic, ISBN 0-7695-1894-X, pp 104111, IEEE, June 2003.

[8] Institute of Electrical and Electronic Engineers, Inc., IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std 754-1985, Aug. 1985.

[9] IEEE Standard for Floating-Point Arithmetic, ANSI/IEEE Std 754-2008, Aug. 2008.

[10] F. Busaba, C. Krygowski, W. Li, E. Schwarz, and S. Carlough, The IBM z900 Decimal Arithmetic Unit, Conference Record of the 35th Asilomar Conference on Signals, Systems and Computers, Vol. 2, ISBN 0 7803 7147 X, pp13351339, IEEE, Nov. 2001.

[11] M. Erle and M. Schulte, Decimal Multiplication Via Carry-Save Addition, in IEEE International Conference on Application-Specific Systems, Architectures, and Processors, I.C.S. Press, ed., the Hague, Netherlands, June 2003, pp 348358.

[12] T. Ueda, Decimal Multiplying Assembly and Multiply Module, U.S. Patent No. 5379245, Jan 1995.

[13] G. Jaberipur and A. Kaivani, Binary-Coded Decimal Digit Multipliers, in IET Computers and Digital Techniques, 2007, pp. 377-381.

[14] M. Erle, E. Schwarz, and M. Schulte, Decimal Multiplication With Efficient Partial Product Generation, in 17th IEEE Symposium on Computer Arithmetic, June 2005, pp2128.

[15] T. Ohtsuki et al., apparatus for decimal multiplication, U.S. Patent No. 4677583, (1987).

[16] T. Lang and A. Nannarelli, A Radix-10 Combinational Multiplier, in 40th Asilomar Conference on Signals, Systems, and Computers, Oct. 2006, pp313317.

[17] A. Vazquez and E. Antelo, Conditional Speculative Decimal Addition, in 7^{th} Conference on Real Numbers and Computers (RNC 7), July 2006, pp. 47-57.

[18] F. Busaba, T. Slegel, S. Carlough, C. Krygowski, and J. Rell,The Design of the Fixed Point Unit for the z990 Microprocessor, in Proceedings of the 14th ACM Great Lakes symposium on VLSI, Apr. 2004, pp. 364 367.

[19] M. D. Ercegovac and T. Lang. Digital Arithmetic. Morgan Kaufmann, 2003.

[20] M. D. Ercegovac, "'On-Line Arithmetic: An Overview"', SPIE, vol. 495, Real Time Signal Processing VII, 1984, pp. 86-93.

[21] K. Trivedi, M. Ercegovac, "'On-Line Algorithms for Division and multiplication"', IEEE Transaction on Computers, vol. c-26, No. 7,July 1977, pp. 681-687.

[22] M. Ercegovac, T. Lang, "'On-line arithmetic for DSP applications,"' 32nd Midwest Symposium on Circuits and Systems, 1989, pp. 365368.

[23] M. Cowlishaw, Densely Packed Decimal Encoding, IEE Proceedings Computers and Digital Techniques, 149(2002), pp. 102-104.

[24] CHEN, T.C., and HO, T.: "'Storage-efficient representation of decimal data"', CACM, 1975, 18, (1), pp. 4952 (summarised at http://www2.hursley.ibm.com/decimal/chen-ho.html)

[25] SMITH, A.J.: 'Comments on a Paper by T.C. Chen and I.T. Ho', CACM,1975, 18, (8), p. 463

[26] A. Svoboda, "'Decimal Adder with Signed Digit Arithmetic,"' IEEE Transaction on Computers, vol. C, pp. 212215, March 1969.

[27] Moskal, J.; Oruklu, E.; Saniie, J.; , "'Design and Synthesis of a Carry-Free Signed-Digit Decimal Adder,"' Circuits and Systems, 2007. ISCAS 2007. IEEE International Symposium on , vol., no., pp.1089-1092, 27-30 May 2007.

[28] B. Shirazi, D. Yun, and C. Zhang, "'RBCD: redundant binary coded decimal adder,— in Computers and Digital Techniques, IEE Proceedings E, vol. 136, no. 2, Mar 1989, pp. 156160.

[29] Rebacz, J.; Oruklu, E.; Saniie, J.; , "'High performance signed-digit decimal adders,"' Electro/Information Technology, 2009. eit '09. IEEE International Conference on , vol., no., pp.251-255, 7-9 June 2009

Vita

- Abdulaziz Salah Tabbakh.
- Nationality: Saudi
- Born in Jeddah, Saudi Arabia, on April 02, 1985.
- Completed Bachelor of Science in Computer Engineering from King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia, in June 2007.
- Present and Permanent Address: Al-Rabwah Dist., Abdullah Bin Ka'ab Street, Jeddah, Saudi Arabia
- Mobile: +966503012953
- Email: azizsn@gmail.com
- Email: atabakh@kfupm.edu.sa.